# Contributions à l'analyse de systèmes par approximation d'ensembles réguliers

Roméo Courbis

Année 2011

# THÈSE

**présentée à**

L'U.F.R. DES SCIENCES ET TECHNIQUES
DE L'UNIVERSITÉ DE FRANCHE-COMTÉ

**pour obtenir**

**LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE FRANCHE-COMTÉ**

**Spécialité : Informatique**

# Contributions à l'analyse de systèmes par approximation d'ensembles réguliers


par

**Roméo Courbis**


**Soutenue le 15 septembre 2011 devant la Commission d'Examen :**

| | | |
|---|---|---|
| **Président du Jury** | Béatrice Bérard | Professeur, Université Pierre & Marie Curie (LIP6), Paris 6 |
| **Directeurs de Thèse** | Olga Kouchnarenko | Professeur, Université de Franche-Comté (LIFC) |
| | Pierre-Cyrille Héam | Professeur, Université de Franche-Comté (LIFC) |
| **Rapporteurs** | Thomas Genet | Maître de Conférence HDR, Université de Rennes 1 (ISTIC/IRISA) |
| | Sébastien Limet | Professeur, Université d'Orléans (LIFO) |
| **Examinateur** | Alain Giorgetti | Maître de Conférence, Université de Franche-Comté (LIFC) |

# Remerciements

Je veux tout d'abord remercier mes deux encadrants, Olga Kouchnarenko et Pierre-Cyrille Héam, pour leur conseils et leur patience durant ces quatre années de thèse.

Je remercie aussi Béatrice Bérard pour avoir accepté les rôles d'examinateur et de président du jury.

Je tiens aussi à remercier Thomas Genet et Sébastien Limet pour avoir accepté d'être les rapporteurs de cette thèse.

Je remercie également Alain Giorgetti pour avoir accepté d'être membre de jury en temps qu'examinateur.

Mes remerciements vont aussi aux membres du LIFC pour leur accueil avant, pendant et après la thèse ; aux membres du projet RAVAJ pour la toujours présente bonne ambiance de travail ; ainsi qu'à l'équipe CASSIS.

Et bien évidement je remercie mes parents, ma famille et mes amis pour leur soutien inconditionnel durant la thèse !

# Table des matières

# Table des figures

# Première partie

# Introduction

# 1 Prologue

**Sommaire**



## 1.1 Contexte

L'informatique tient une place de plus en plus importante aussi bien dans le domaine de notre vie courante que dans de nombreux domaines spécifiques : ordinateurs, téléphones portables, ordinateurs de bord (voitures, avions, fusées, satellites, ...), centrales nucléaires, et bien d'autres. Ces domaines intègrent des systèmes informatiques dont les enjeux peuvent aller de la gestion de notre carnet d'adresses, de notre classement à un jeu de go en ligne, à la gestion de transactions bancaires via des réseaux ou au calcul de la position d'une fusée.

Dans tous ces cas de figure, un bogue ou une erreur de conception peut provoquer des résultats inattendus ou non désirés. Par exemple, le logiciel de navigation d'un F-16 provoquait le retournement de l'avion au passage à l'équateur (à cause d'une inversion de signe). Ou encore, sur une des premières version de l'avion, l'ordinateur de bord permettait la rétraction des trains d'atterrissage alors que l'avion était encore au sol.

Ces deux exemples montrent que des vies humaines et d'importantes sommes d'argent peuvent être dépendantes de systèmes informatiques, d'où l'importance de l'évaluation de la qualité de ceux-ci.

On peut distinguer deux techniques pour évaluer la qualité d'un système informatique : le *test* et la *vérification*.

Le *test* d'un système informatique consiste à faire réagir ce système par rapport à des cas d'utilisation. Les réactions obtenues sont comparées aux attentes définies par la spécification du système. En général, tous les cas d'utilisation ne sont pas testés car leur nombre peut être trop important ou le niveau de qualité requis pour le système testé ne nécessite pas de tests exhaustifs par exemple. Ainsi le test ne garantit pas qu'un système informatique fonctionne toujours comme défini dans sa spécification. De nombreux travaux ont été réalisés pour améliorer l'efficacité et l'automatisation de cette méthode.

La *vérification* permet de prouver qu'un modèle du système informatique répond à toutes les attentes (ou propriétés) décrites dans sa spécification. La vérification peut se faire avec un assistant de preuve permettant de répondre semi-automatiquement à toutes les questions de vérification. Il existe des logiciels permettant d'assister un ingénieur pour la réalisation de ces démonstra-

tions. La vérification peut aussi se faire par évaluation de modèle, model-checking [CES83, QS82] en anglais. C'est une méthode principalement automatique permettant de vérifier qu'un modèle du système informatique satisfait ou non des propriétés pour tous les cas d'utilisation possibles de ce système.

Il est possible de discerner différents types de propriétés. L'intérêt de cette différenciation réside dans le fait que suivant la modélisation choisie, certains types de propriétés sont plus ou moins difficiles, voire impossible, à vérifier. Plusieurs distinctions sont possibles, une première a été introduite dans [Lam77] entre les propriétés de sûreté et de vivacité. Ici nous allons utiliser une classification courante, que l'on peut retrouver dans [SBB+99] :

- Les *propriétés de sûreté* énoncent que quelque chose de mauvais n'arrivera jamais, sous certaines conditions. Par exemple, *les trains d'atterrissage d'un avion ne peuvent pas être rétractés si l'avion est au sol.*
- Les *propriétés d'atteignabilité* modélisent des états qui peuvent être atteints par le système. Par exemple, *les trains d'atterrissage d'un avion peuvent être rétractés.*
- Les *propriétés de vivacité* expriment que quelque chose arrivera, sous certaines conditions. Par exemple, *si un avion a décollé, il finira par toucher le sol un jour.*
- Et bien d'autres comme les *propriétés d'interblocage* (ou deadlock), *d'équité*, ...

Le model-checking nécessite la modélisation du système et des propriétés à vérifier à l'aide de langages spécifiques. Ensuite ces deux données sont fournies à un logiciel, appelé model-checker, permettant de déterminer si la propriété donnée est vraie pour le modèle du système ou non. Dans ce dernier cas, un model-checker peut donner un exemple de cas d'utilisation qui contredit la propriété.

Cette thèse se place dans le cadre du model-checking, et plus précisément dans le cadre du model-checking régulier [BG96, BW98, BJNT00].

## 1.2 Le model-checking régulier

Le model-checking régulier [BG96, BW98, BJNT00] est un ensemble de techniques utilisées pour la vérification automatique de propriétés de sûreté et d'atteignabilité pour des systèmes infinis, comme les systèmes comprenant des files FIFO non bornées, des entiers ou des horloges par exemple.

Le point commun de ces techniques réside dans la manière de modéliser les systèmes :

- Les états initiaux du système sont représentés par un ensemble régulier de mots ou de termes $I$.
- L'évolution du système est représentée par une relation $R$.
- La propriété à vérifier est représentée par un ensemble régulier de mots ou de termes $P$.

Le problème, généralement indécidable, du model-checking régulier peut s'écrire de la manière suivante :

$$R^*(I) \cap P = \emptyset \ ?$$

(où $R^*(I)$ est l'image du langage régulier $I$ par la clôture transitive et réflexive de la relation $R$.)

Le model-checking régulier permet de réaliser une analyse d'accessibilité régulière : à partir d'un ensemble d'états initiaux régulier $I$, une relation d'évolution $R$ et un ensemble d'états $P$, est-ce qu'il est possible d'accéder à $P$ à partir des états initiaux $I$ en respectant la relation d'évolution $R$ ?

Comme la figure 1.1 le montre, l'ensemble $R^*(I)$ représente l'ensemble des états du système analysé et l'analyse d'accessibilité se fait en calculant l'intersection entre les deux ensembles

$R^*(I)$ et $P$.

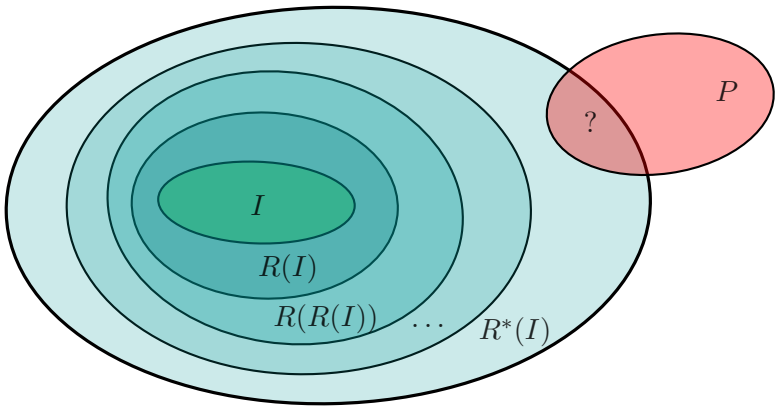



FIGURE 1.1 – Analyse d'accessibilité régulière

Cependant l'ensemble $R^*(I)$ n'est pas régulier (ni même calculable) en général. Par exemple, si on a le terme $t = f(a, a)$ et le système de réécriture $R = \{f(x, y) \to f(g(x), g(y))\}$, on a l'ensemble non-régulier $R^*(\{t\}) = \{f(g^n(a), g^n(a)) \mid n \in \mathbb{N}\}$. En effet, on peut voir qu'il est possible de réécrire une infinité de fois le terme $t$ à l'aide du système de réécriture $R$.

Il existe trois approches pour aborder le problème du model-checking régulier :

- trouver des sous-classes décidables de ce problème ;
- calculer une approximation de l'ensemble $R^*(I)$ ;
- décomposer la relation $R$ en un nombre fini de sous-relations $R_i$ (telles que $R = \cup_i R_i$) afin de réaliser des accélérations calculant en une étape une infinité d'éléments de $R_i^*(I)$.

**Sous-classes décidables.** Pour le cas où le problème du model-checking (ou d'accessibilité) est décidable, on peut citer les systèmes de réécriture terminant où il est possible de calculer l'ensemble des termes atteignables. On peut aussi citer les travaux de [FGT04, GV98, TKS00] où un automate d'arbres représentant un ensemble, potentiellement infini, de termes atteignables est calculé à partir des termes initiaux. Ensuite il est possible de savoir si certains termes sont atteignables ou non.

**Approches par approximations.** L'approche par approximation permet d'avoir une approximation de l'ensemble $R^*(I)$ des termes atteignables.

Dans [BHV04a], les auteurs utilisent des transducteurs et des automates de mots pour calculer une abstraction régulière. Ils étendent cette technique aux automates d'arbres dans [BHRV06].

Dans [Tak04], l'auteur utilise l'interprétation abstraite afin de calculer une abstraction de $R^*(I)$, où $R$ est un système de réécriture et $I$ un langage régulier d'arbre.

Dans cette thèse nous allons nous intéresser principalement à la méthode de complétion d'automate d'arbres [FGT04, Gen98], qui permet le calcul d'une sur-approximation ou d'une sous-approximation de l'ensemble des termes atteignables. Ce calcul se fait à partir d'un système de réécriture et d'un automate d'arbres et est paramétré par une fonction d'approximation définie à l'aide d'un langage particulier et définissant des fusions d'états. Cette technique a été précédemment utilisée avec succès pour la vérification de protocoles cryptographiques [GK00, GTTVTT03, ABB+05, BHK06, BHK08], ou pour la vérification de programmes écrits en bytecode Java [BGJR07].

**Réalisation d'accélérations.** Le relation $R$ est décomposée en $k$ sous-relations telles que $R = \bigcup_{0<i\leq k} R_i$. Ensuite des algorithmes sont utilisés pour calculer des accélérations $T_i$ déterminant (si possible) en une seule étape de calcul une infinité d'éléments de $R_i^*(I)$, où $I$ est un langage régulier. Autrement dit, il faut que $R_i(I) \subseteq T_i(I) \subseteq R_i^*(I)$ ou que $T_i(I) = R_i^*(I)$ si possible. Le calcul du point fixe est réalisé en utilisant ces accélérations ; cependant cette méthode peut ne pas terminer, elle est semi-algorithmique.

Dans cette thèse nous allons nous intéresser à la résolution du problème du model-checking régulier à l'aide d'une méthode calculant une sur-approximation $K$ de l'ensemble $R^*(I)$. L'objectif est d'obtenir si possible un ensemble $K$ tel que son intersection avec $P$ est vide afin de montrer que les termes de l'ensemble $P$ ne sont pas atteignables (figure 1.2). Notons que cela n'est pas toujours possible [BH08b].

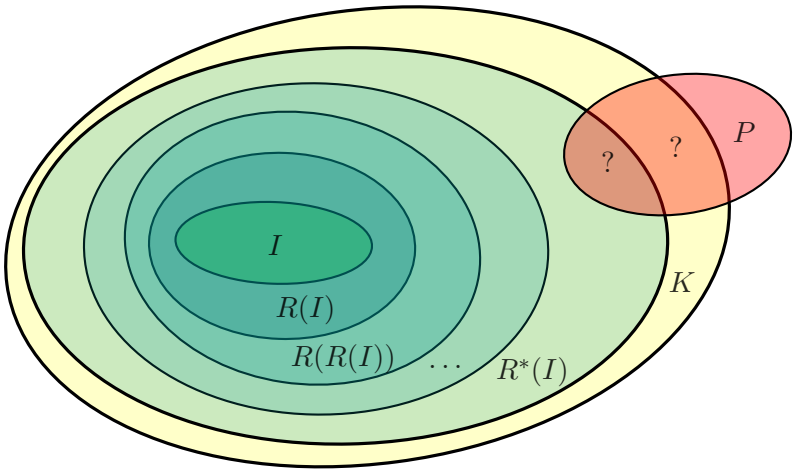



FIGURE 1.2 – Le model-checking régulier par sur-approximation

## 1.3 Contributions

L'objectif de cette thèse est d'explorer d'autres domaines d'applications de la méthode exposée dans [FGT04, Gen98], de la modifier afin de faciliter et d'étendre ses possibilités d'utilisation.

### Application à d'autres domaines

Nous verrons deux autres applications de l'analyse d'atteignabilité par réécriture d'approximation : la vérification de spécifications écrites avec l'algèbre de processus CCS [Mil89] et l'analyse d'atteignabilité par approximation de deux problèmes indécidables pour les machines de Turing.

Pour la première application, nous présentons une méthode de traduction automatique de la sémantique de CCS (sans renommage) et de la spécification CCS à vérifier. Il est ensuite possible de réaliser une analyse d'accessibilité des séquences d'actions réalisables par une spécification CCS.

Pour la deuxième, nous exploitons l'analyse d'atteignabilité pour deux problèmes indécidables pour les machines de Turing : vacuité d'un langage et appartenance d'un mot à un langage. Nous proposons une modélisation et montrons comment les approximations par réécriture permettent de semi-décider ces problèmes.

Dans ces deux cas, l'approximation a été définie par les utilisateurs et non automatiquement. La définition de l'approximation nécessite une connaissance précise du système analysé et de la procédure de complétion. Une mauvaise définition de l'approximation mène à une analyse inconclusive, soit parce que l'approximation est trop grossière et les termes indésirables appartiennent à l'approximation (uniquement vrai pour une sur-approximation), soit l'approximation est trop faible et dans ce cas la procédure de complétion ne termine pas.

### Raffiner les approximations

On voudrait s'affranchir de l'expertise et du temps nécessaire à un opérateur humain pour définir une approximation pertinente — par rapport aux propriétés à vérifier — et automatiser la procédure de complétion.

C'est dans ce cadre que se place une des contributions de cette thèse : le raffinement d'approximation, s'inspirant du paradigme CEGAR (*Counterexample-Guided Abstraction Refinement*) exposé dans [Cla03]. L'objectif du raffinement d'approximation est de construire automatiquement une approximation qui ne contient pas de terme indésirable (représentant la propriété). Si cette construction échoue alors des termes indésirables sont atteignables et on peut conclure que le système analysé ne vérifie pas la propriété. Pour cette technique, un prototype a été implémenté à l'aide de l'outil Timbuk [GVTT01] et quelques expérimentations ont été réalisées afin de montrer l'intérêt du raffinement d'approximation.

### Extension aux systèmes de réécriture non-linéaire à gauche

De plus, la procédure de complétion nécessite un système de réécriture linéaire à gauche ou un automate d'arbres déterministe. Cette procédure effectue le calcul de l'approximation étape par étape jusqu'à atteindre un point fixe et chaque étape ne préserve pas le déterminisme de l'automate d'arbres complété. Si on se place dans le cas où on a un système de réécriture non linéaire à gauche, il faut donc forcément avoir un automate d'arbres déterministe à chaque étape. Cependant, comme la complétion ne préserve pas le déterminisme, il n'est pas raisonnable de déterminiser un automate d'arbres à chaque étape de complétion (car chaque étape de déterminisation peut potentiellement faire exploser le nombre d'états de l'automate).

L'objet d'une contribution de cette thèse est la définition d'une approche algorithmique permettant d'éviter l'opération de déterminisation à chaque étape, qui est une opération de complexité exponentielle, et de la remplacer par une opération de complexité polynomiale.

### Vérifier des propriétés temporelles par analyse d'accessibilité

La méthode de vérification par réécriture d'approximation utilisant la complétion utilise des automates d'arbres pour modéliser les propriétés à vérifier.

Dans une autre contribution, nous présentons comment vérifier trois modèles de propriétés LTL à l'aide de la complétion :

- $\Box(R_1 \Rightarrow \circ R_2)$,
- $\neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$,
- $\Box(R_1 \Rightarrow \Box\neg R_2)$.

Ces propriétés expriment des conditions sur l'ordre d'application des règles de réécriture lors de la réécriture d'un terme de l'ensemble des configurations initiales. L'objectif est de déterminer si oui ou non un modèle vérifie ces propriétés. Ce modèle est un graphe dont les états sont des termes et les transitions sont étiquetées par un ensemble de règles de réécriture. Il représente toutes les réécritures possibles d'un ensemble de termes reconnu par un automate d'arbres par un système de réécriture $R$ tel que $R_1$, $R_2 \in R$.

## 1.4 Plan

Dans le chapitre 2, nous introduisons toutes les définitions, notations, propositions et théorèmes indispensables à la lecture de cette thèse. Nous rappelons tout d'abord les notions de termes, de systèmes de réécriture et d'automates d'arbres. Ensuite nous présentons la procédure de complétion, qui est à la base de cette thèse.

La partie II comprend des applications de la réécriture par complétion à deux problèmes de vérifications. La première concerne la vérification de spécifications CCS (chapitre 3). La deuxième concerne la résolution de deux problèmes indécidables pour les machines de Turing (chapitre 4), la décision du vide et de l'appartenance d'un mot.

La partie III contient les principales contributions apportées par cette thèse au model-checking régulier par réécriture et par approximation. Dans le chapitre 6, nous présentons une méthode de raffinement d'approximation utilisant la complétion, suivant le paradigme CEGAR [Cla03]. Le chapitre 7 présente une méthode permettant d'appliquer la complétion lorsque l'on a un système de réécriture non-linéaire à gauche. Dans le chapitre 8, on montre comment vérifier trois modèles de propriétés LTL en s'aidant de la procédure de complétion et des automates avec contraintes.

## 1.5 Publications

Cette thèse s'appuie sur des travaux ayant fait l'objet de publications. Voici une liste de ces publications auxquelles j'ai participé :

**[BCHK08a]** traite du raffinement d'approximation et est l'objet du chapitre 6.

**[BCHK08b]** présente une méthode permettant d'appliquer la procédure de complétion à des systèmes de réécriture quadratiques à gauche.

**[BCHK09]** étend la méthode exposée dans [BCHK08b] aux systèmes de réécriture non linéaires à gauche, méthode présentée au chapitre 7.

**[CHK09]** traite de la vérification de trois propriétés LTL à l'aide de la réécriture d'approximation (voir le chapitre 8).

**[CHJK10]** présente une modélisation des machines de Turing sous forme de système de réécriture et d'automate d'arbres afin d'aborder deux problème indécidables : le problème de vacuité et de la reconnaissance d'un mot par une machine de Turing (voir le chapitre 4).

**[CCF+10]** expose une façon de paralléliser et de distribuer la procédure de complétion.

**[Cou11]** traite de la vérification de spécifications CCS à l'aide de la réécriture d'approximations (voir chapitre 3).

# 2

# Préliminaires théoriques

**Sommaire**



Ce chapitre comprend toutes les définitions, propositions, théorèmes indispensables à la compréhension de la suite de ce document.

Nous présentons tout d'abord les termes (section 2.1), puis les systèmes de réécriture (section 2.2) et les automates d'arbres bottom-up et à contraintes d'égalité et de différence (section 2.3). Ces notions nous permettent ensuite de présenter la méthode de réécriture par complétion (section 2.4).

**Entiers naturels et mots.**

On note $\mathbb{N}$ l'ensemble des entiers naturels. On note $\mathbb{N}^*$ l'ensemble des mots sur $\mathbb{N}$ et $\epsilon$ le mot vide. On note $|m|$ la taille du mot $m$.

## 2.1 Termes

On note $\mathcal{F}$ un ensemble de symboles fonctionnels et $Arity$ une fonction de $\mathcal{F}$ dans $\mathbb{N}$. On dit que $\mathcal{F}_n$ est l'ensemble des symboles d'arité $n$ :

$$\mathcal{F}_n = \{f \in \mathcal{F} \mid Arity(f) = n\}.$$

Lors de la définition d'un ensemble de symboles $\mathcal{F}$, l'arité des symboles pourra être notée en exposant de chaque symbole. Par exemple pour l'ensemble $\mathcal{F} = \{f^2, a^0\}$, $f$ est d'arité 2 et $a$ est d'arité 0.

Soit $\mathcal{X}$ un ensemble fini de variables tel que $\mathcal{F}$ est disjoint de $\mathcal{X}$. On note $\mathcal{T}(\mathcal{F}, \mathcal{X})$ le plus petit ensemble des termes ouverts $t$ défini inductivement par :

- $t = f(t_1, \ldots, t_n)$ avec $n > 1$, $f \in \mathcal{F}_n$ et $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- ou $t \in \mathcal{F}_0$,
- ou $t \in \mathcal{X}$.

Lorsque $\mathcal{X}$ est l'ensemble vide alors $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est noté $\mathcal{T}(\mathcal{F})$, que l'on appelle l'ensemble des termes clos.

On note $C[q_1, \ldots, q_n]$ un terme de $\mathcal{T}(\mathcal{F} \cup \{q_1, \ldots, q_n\})$, où $q_1, \ldots, q_n$ sont des symboles de $\mathcal{F} \cup \mathcal{X}$.

♢ **Exemple 2.1**
*Soit* $\mathcal{F} = \{a^0, b^0, g^1, f^2\}$ *et* $\mathcal{X} = \{x, y\}$. *Le terme* $f(a, g(b))$ *appartient à* $\mathcal{T}(\mathcal{F})$, *le terme* $f(x, g(f(y, a)))$ *appartient à* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *et* $f(x)$ *n'est pas un terme de* $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

On peut représenter graphiquement un terme sous la forme d'un arbre, illustré par la figure 2.1. Les nœuds sont des symboles, et le nombre de fils d'un nœud correspond à l'arité du symbole correspondant à ce nœud.

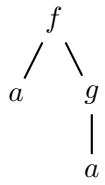



FIGURE 2.1 – Représentation du terme $f(a, g(b))$.

On note $\mathcal{V}ar(t)$ l'ensemble des variables d'un terme $t$ de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ défini inductivement par :
- si $t = f(t_1, \ldots, t_n)$ avec $n > 1$, $f \in \mathcal{F}_n$ et $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ alors $\mathcal{V}ar(t) = \bigcup_{1 \leq i \leq n} \mathcal{V}ar(t_i)$
- et si $t \in \mathcal{F}_0$ alors $\mathcal{V}ar(t) = \emptyset$
- et si $t \in \mathcal{X}$ alors $\mathcal{V}ar(t) = \{t\}$.

On dit qu'un terme $t$ est linéaire quand chaque variable de $\mathcal{V}ar(t)$ n'apparaît qu'une seule fois (au plus) dans $t$. C'est-à-dire qu'un terme $t$ de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est linéaire si :
- $t \in \mathcal{F}_0$ ou $t \in \mathcal{X}$ ou
- $t = f(t_1, \ldots, t_n)$, avec $n > 1$, $f \in \mathcal{F}_n$ et $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, on a $\bigcap_{1 \leq i \leq n} \mathcal{V}ar(t_i) = \emptyset$ et les termes $t_1, \ldots, t_n$ sont linéaires.

Un terme dans lequel chaque variable apparaît au plus $h$ fois est dit $h$-linéaire.

♢ **Exemple 2.2**
*Soit* $t = f(a, g(x, y))$ *et* $s = f(a, g(x, x))$ *des termes ouverts tels que* $x, y \in \mathcal{X}$ *et* $f^2,\ g^2,\ a^1 \in \mathcal{F}$. *On a* $\mathcal{V}ar(t) = \{x,\ y\}$ *et* $\mathcal{V}ar(s) = \{x\}$. *Le terme* $t$ *est linéaire contrairement au terme* $s$ *qui est* 2*-linéaire.*

Maintenant nous allons définir la notion de position d'un terme.

Pour un terme $t$, on définit $\mathcal{P}os(t) \subseteq \mathbb{N}^*$, l'ensemble des positions de $t$, comme suit :
- si $t \in \mathcal{F}_0$ ou $t \in \mathcal{X}$ alors $\mathcal{P}os(t) = \{\epsilon\}$,
- si $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, avec $t = f(t_1, \ldots, t_n)$, alors $\mathcal{P}os(t) = \{\epsilon\} \cup \{1.p_1, \ldots, n.p_n \mid p_1 \in \mathcal{P}os(t_1), \ldots, p_n \in \mathcal{P}os(t_n)\}$.

Chaque élément de $\mathcal{P}os(t)$ est appelé une position. On note $t|_p$ le sous-terme de $t$ à la position $p$ défini par :
- $t|_\epsilon = t$,
- si $t = f(t_1, \ldots, t_n)$ alors $t|_{i.p'} = t_i|_{p'}$ $(1 \leq i \leq n)$.

On définit le remplacement du sous-terme de $t$ à la position $p$ par le terme $s$, noté $t[s]_p$, par :
- Si $p = \epsilon$, alors $t[s]_\epsilon = s$,
- Si $p = i.p'$ avec $i \in \mathbb{N}$ et si $t = f(t_1, \ldots, t_n)$ alors $t[s]_{i.p'} = f(t_1, \ldots, t_i[s]_{p'}, \ldots, f_n)$ $(1 \leq i \leq n)$.

On note $\mathcal{P}os_M(t)$ l'ensemble des positions des termes appartenant à $M$ dans $t$ :

$$\mathcal{P}os_M(t) = \{p \in \mathcal{P}os(t) \mid t|_p \in M\}$$

♢ **Exemple 2.3**
*Soit* $t = f(a, g(a, b))$ *et* $s = h(x)$. *On a :*
$\mathcal{P}os(t) = \{\epsilon,\ 1,\ 2,\ 2.1,\ 2.2\}$
$\mathcal{P}os_{\mathcal{X}}(s) = \{1\}$
$t|_2 = g(a, b)$
$t[s]_2 = f(a, h(x))$
$t[s]_{2.1} = f(a, g(h(x), b))$
$t[s]_1[s]_{2.1} = f(h(x), g(h(x), b))$

Si $t$ est un terme et $p$ une position de $t$, $t(p)$ désigne le symbole apparaissant en position $p$ dans $t$. Plus formellement,

- Si $t \in \mathcal{F}_0$ ou $t \in \mathcal{X}$, alors $t(\epsilon) = t$.
- Si $t = f(t_1, \ldots, t_n)$, avec $n > 1$, $f \in \mathcal{F}_n$, alors $t(\epsilon) = f$.
- Si $p = i.p'$ avec $i \in \mathbb{N}$ et si $t = f(t_1, \ldots, t_n)$, avec $n > 1$, $f \in \mathcal{F}_n$, alors $t(p) = t_i(p')$.

Grâce à ces définitions nous allons pouvoir définir la notion de substitution. Une substitution $\sigma$ est une application de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F})$. Si on prend un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $t\sigma$ représente le terme où chaque variable $x \in \mathcal{V}ar(t)$ a été remplacée par le terme $\sigma(x)$ :

$$t\sigma = t[\sigma(x_1)]_{p_1} \ldots [\sigma(x_n)]_{p_n}$$

avec $\mathcal{P}os_{\mathcal{X}}(t) = \{p_1, \ldots, p_n\}$ et $x_1 = t|_{p_1}, \ldots, x_n = t|_{p_n}$.

♢ **Exemple 2.4**
*Soit* $t = f(a, g(x, y))$ *avec* $f,\ g \in \mathcal{F}_2$, $a \in \mathcal{F}_0$ *et* $x,\ y \in \mathcal{X}$. *Soit* $\sigma$ *une substitution telle que* $\sigma = \{x \to h(x),\ y \to a\}$, *on a :*

$$t\sigma = f(a, g(h(x), a)).$$

Les notions de remplacement d'un sous-terme et de substitutions nous permettent de réécrire un terme en un autre comme on a pu le voir dans les exemples précédents. Les réécritures de termes peuvent obéir à des règles de réécriture. Celles-ci sont constituées en systèmes de réécriture que nous présentons dans la section suivante.

## 2.2 Systèmes de réécriture

Une règle de réécriture est un couple $(l, r)$ que l'on notera $l \to r$ avec $l,\ r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Le terme $l$ est appelé la partie gauche de la règle $l \to r$, et $r$ la partie droite. Un système de réécriture est un ensemble fini de règles de réécriture.

On dit qu'un système de réécriture $R$ est linéaire à gauche (respectivement à droite) si pour chacune des règles $l \to r$ de $R$, $l$ est linéaire (respectivement $r$ est linéaire). Un système de réécriture est linéaire quand il est linéaire à gauche et à droite.

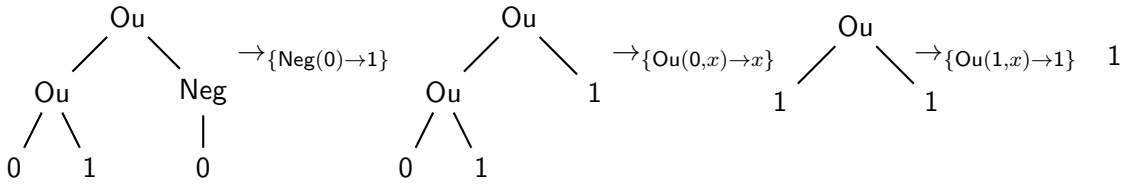
♢ **Exemple 2.5** (Encodage du "ou" logique et de la négation en système de réécriture)
*Soit* $\mathcal{F} = \{1^0, 0^0, \mathsf{Neg}^1, \mathsf{Ou}^2\}$ *un ensemble de symboles et* $\mathcal{X} = \{x\}$ *un ensemble de variables. Soit* $R_\vee = \{\mathsf{Ou}(1, x) \to 1, \mathsf{Ou}(0, x) \to x, \mathsf{Neg}(0) \to 1, \mathsf{Neg}(1) \to 0, \mathsf{Neg}(\mathsf{Neg}(x)) \to x\}$ *un système de réécriture. Les règles de* $R_\vee$ *sont linéaires, le système de réécriture* $R_\vee$ *est linéaire.*

*Si* $R_\vee$ *contenait la règle de réécriture* $\mathsf{Ou}(x, x) \to x$ *(linéaire à droite mais pas à gauche), alors ce système de réécriture ne serait plus linéaire à gauche.*

Pour un terme $t$ et une règle de réécriture $l \to r$, la réécriture de $t$ par $l \to r$ consiste à chercher une position $p \in \mathcal{P}os(t)$ et une substitution $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ telles que $t|_p = l\sigma$ puis à remplacer $l\sigma$ par $r\sigma$ afin d'obtenir le terme $t[r\sigma]_p$. Ceci nous amène à définir une relation de réécriture.

△ **Définition 2.6** (Relation de réécriture)
*Soit deux termes $t$, $s$ dans $\mathcal{T}(\mathcal{F})$ et une règle de réécriture $l \to r$ d'un système de réécriture $R$. Le terme $t$ se réécrit en un terme $s$ par la règle $l \to r$, noté $t \to_R s$, s'il existe une position $p$ de $\mathcal{P}os(t)$ et une substitution $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ telles que $t = t[l\sigma]_p$ et $s = t[r\sigma]_p$.*

♢ **Exemple 2.7** (Évaluation de $(0 \vee 1) \vee (\neg 0)$ à l'aide du système de réécriture $R_\vee$)
*On montre ici la réécriture possible d'un terme.*




De plus, pour un ensemble de termes $E$, on notera $R^{-1}(E)$ l'ensemble de termes $\{s \mid \exists t \in E$ tel que $s \to_R t\}$.

La clôture transitive et réflexive de $\to_R$ est notée $\to_R^*$.

On dit que $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ est *réductible* par $R$ si il existe un terme $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ tel que $t \to_R s$. Un terme $t$ est en *forme normale* s'il n'est pas réductible.

♢ **Exemple 2.8**
*Pour le système de réécriture $R_\vee$, on a :*
- *1 est en forme normale,*
- *$\mathsf{Ou}(1,1)$ est réductible par $R_\vee$,*
- *$f(a)$ est en forme normale.*

On dit qu'un système de réécriture $R$ est *terminant* si pour tout terme $t$ il n'existe pas de suite infinie $t \to_R t_1 \to_R t_2 \to_R \ldots$ de réécritures (où $t_1$, $t_2$, … sont des termes).

♢ **Exemple 2.9**
*Le système de réécriture $R_\vee$ est terminant car de manière intuitive on peut voir que chacune des règles permet de réécrire un terme en un terme comprenant moins de positions.*

*Par contre le système de réécriture $R'_\vee = R_\vee \cup \{\mathsf{Ou}(x,y) \to \mathsf{Ou}(y,x)\}$ n'est pas terminant car le terme $\mathsf{Ou}(0,1)$ peut être réécrit un nombre infini de fois par la règle $\mathsf{Ou}(x,y) \to \mathsf{Ou}(y,x)$.*

Un système de réécriture $R$ est *confluent* si pour tous termes $s$, $t$, $u$ tels que $s \to_R^* t$ et $s \to_R^* u$ il existe un terme $v$ tel que $t \to_R^* v$ et $u \to_R^* v$ (figure 2.2).

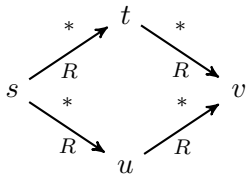



FIGURE 2.2 – La propriété de confluence.

♢ **Exemple 2.10**
*Les systèmes de réécriture $R_\vee$ et $R'_\vee$ sont confluents car toute évaluation d'une même expression logique donne le même résultat (en supposant qu'il représente l'évaluation d'une expression contenant des ou et des négations).*

*Le système de réécriture $R_1 = \{\mathsf{choix}(x,y) \to x,\ \mathsf{choix}(x, g(y)) \to g(y)\}$ n'est pas confluent : le terme $\mathsf{choix}(a, g(b))$ peut se réécrire soit en $a$ soit en $g(b)$ qui sont deux termes en forme normale.*

Nous allons maintenant nous intéresser aux automates d'arbre, dernière notion indispensable pour la compréhension de la technique de complétion décrite dans la section 2.4.

## 2.3 Automates d'arbres ...

Dans cette section nous allons voir deux types d'automates d'arbres : les automates d'arbres bottom-up et les automates d'arbres à contraintes d'égalité et de différence qui sont une extensions des automates d'arbres bottom-up. Seuls les automates d'arbres de type bottom-up sont indispensables à la compréhension de la section 2.4.

### 2.3.1 ... bottom-up

Un automate d'arbres *bottom-up* est un quadruplet $(\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ où $\mathcal{Q}$ est un ensemble fini de symboles d'arité 0 que l'on appelle états (avec $\mathcal{Q} \cap \mathcal{F} = \emptyset$), $\mathcal{Q}_f$ est l'ensemble des états finaux (avec $\mathcal{Q}_f \subseteq \mathcal{Q}$) et $\Delta$ est un système de réécriture où chaque règle (appelée transition) est du type $t \to q$ avec $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q \in \mathcal{Q}$. Chaque transition de $\Delta$ est de la forme $t \to q$ où $t$ est de la forme $f(q_1, \ldots, q_n)$ avec $f \in \mathcal{F}_n$ et $q, q_1, \ldots, q_n \in \mathcal{Q}$.

Par la suite, nous appellerons automate d'arbres un automate d'arbres *bottom-up*.

♢ **Exemple 2.11**
*Soit $f(a) \to q$ et $f(q_a) \to q$ des règles de réécriture telles que $\mathcal{F} = \{f^2, a^0\}$ et $q_a \in \mathcal{Q}$.*
*La règle $f(a) \to q$ n'est pas une transition d'un automate d'arbres car $a \notin \mathcal{Q}$.*
*La règle $f(q_a) \to q$ est une transition d'un automate d'arbres car $q_a \in \mathcal{Q}$.*

L'ensemble des transitions $\Delta$ d'un automate $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ induit une relation de réécriture que l'on note $\to_A$ ou encore $\to_\Delta$. Si $\Delta$ contient une seule transition $\delta$, on notera cette relation $\to_\delta$.

Un automate d'arbres $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ reconnaît un langage, que l'on note $L(A)$, définit par :

$$L(A) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_\Delta^* q \text{ avec } q \in \mathcal{Q}_f\}.$$

Soit $E \subseteq \mathcal{T}(\mathcal{F})$ un ensemble de termes, on dit que $E$ est un langage *régulier* s'il existe un automate d'arbres $A$ tel que $L(A) = E$.

♢ **Exemple 2.12**
*Soit l'automate $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ tel que $\mathcal{F} = \{f^2, g^2, a^0, b^0\}$, $\mathcal{Q} = \{q_a,\ q_b,\ q_g,\ q_f\}$, $\mathcal{Q}_f = \{q_f,\ q_g\}$ et $\Delta = \{f(q_a, q_g) \to q_f,\ g(q_b, q_a) \to q_g,\ a \to q_a,\ b \to q_b\}$.*

*Le terme $f(a, g(b, a))$ est reconnu par l'automate $A$ car :*

$$f(a, g(b, a)) \to_\Delta f(q_a, g(q_b, q_a)) \to_\Delta f(q_a, q_g) \to_\Delta q_f.$$

*L'automate d'arbres $A$ reconnaît le langage suivant :*

$$L(A) = \{f(a, g(b, a)),\ g(b, a)\}.$$

Pour les langages réguliers les problèmes d'appartenance, d'inclusion et de vacuité sont décidables. L'ensemble des langages réguliers est clos pour les opérations d'intersection, d'union et de différence.

Une *exécution* d'un automate d'arbres $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ sur un terme $t \in \mathcal{T}(\mathcal{F})$ est une application $\rho : \mathcal{P}os(t) \to \mathcal{Q}$ telle que, pour tout $p \in \mathcal{P}os(t)$, si $t|_p = f(t_1, \ldots, t_n)$ $(Arity(f) = n)$, alors il existe $f(\rho(p.1), \ldots, \rho(p.n)) \to q \in \Delta$ telle que $\rho(p) = q$.

Étant donné une exécution $\rho$, elle est *réussie* si $\rho(\epsilon) \in \mathcal{Q}_f$. On montre facilement que le langage reconnu d'un automate d'arbres $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ est l'ensemble de termes $t \in \mathcal{T}(\mathcal{F})$ pour lesquels il existe une exécution réussie de $A$.

Par la suite, on représentera graphiquement une exécution $\rho$ à l'aide de la représentation d'un terme $t$ à laquelle on ajoutera à chaque nœud $\rho(p)$, pour chaque $p \in \mathcal{P}os(t)$.

♢ **Exemple 2.13**
*Soit l'automate* $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$, *de l'exemple précédent, tel que* $\mathcal{F} = \{f^2, g^2, a^0, b^0\}$, $\mathcal{Q} = \{q_a,\ q_b,\ q_g,\ q_f\}$, $\mathcal{Q}_f = \{q_f,\ q_g\}$ *et* $\Delta = \{f(q_a, q_g) \to q_f,\ g(q_b, q_a) \to q_g,\ a \to q_a,\ b \to q_b\}$. *Soient* $t_1$, $t_2$ *et* $t_3$ *des termes de* $\mathcal{T}(\mathcal{F})$, *tels que* $t_1 = g(b, a)$, $t_2 = f(a, g(b, a))$ *et* $t_3 = g(a, a)$.

*On a l'exécution* $\rho_1$ *(resp.* $\rho_2$, $\rho_3$*), de l'automate* $A$ *sur les termes* $t_1$ *(resp.* $t_2$, $t_3$*), suivant :*

- $t_1$ : $\rho_1(1) = q_b$, $\rho_1(2) = q_a$, $\rho_1(\epsilon) = q_g$
- $t_2$ : $\rho_2(2.1) = q_b$, $\rho_2(2.2) = q_a$, $\rho_2(2) = q_g$, $\rho_2(1) = q_a$, $\rho_2(\epsilon) = q_f$
- $t_3$ : $\rho_3(1) = \rho_3(2) = q_a$

*Les exécutions* $\rho_1$ *et* $\rho_2$ *sont des exécutions réussies car* $\rho_1(\epsilon)$, $\rho_2(\epsilon) \in \mathcal{Q}_f$, *contrairement à l'exécution* $\rho_3$ *qui ne l'est pas. L'exécution* $\rho_1$ *est représentée figure 2.3.*

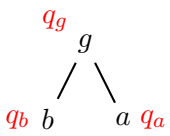



FIGURE 2.3 – Une exécution de $A$ sur $g(b, a)$.

Un automate d'arbres $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ est *déterministe* si pour tout symbole $f$ de $\mathcal{F}$ d'arité $n$ ($n \geq 0$), et pour tous états $q_1, q_2, \ldots, q_n$ de $\mathcal{Q}$ il existe au plus un état $q$ de $\mathcal{Q}$ tel que $f(q_1, q_2, \ldots, q_n) \to q$ est une transition de $\Delta$.

Pour tout automate d'arbres $A_{nd}$, il existe un automate d'arbres déterministe $A_d$ tel que $L(A_{nd}) = L(A_d)$. La déterminisation d'un automate d'arbres est de complexité exponentielle dans le pire des cas.

### 2.3.2 ... avec contraintes d'égalité et de différence

Afin de modéliser plus précisément les systèmes et de fournir de meilleurs approximations, des nouvelles classes étendues d'automate d'arbres ont été définies [BT92, DCC95, CC05, FTT08, SSMH04, KL07, OT05, JRV06].

Les automates d'arbres avec contraintes d'égalité et de différence [FTT08, FTT10], ou TAGED (Tree Automata with Global Equalities and Disequalities), sont des automates d'arbres bottom-up, où des contraintes d'égalité (ou de différence) peuvent être fixées arbitrairement entre les différents sous arbres. Dans le cadre de cette thèse nous allons utiliser uniquement les TAGED munis de contraintes d'égalité, que l'on nommera TAGED positifs (voir aussi [JKV09, JKV11]).

△ **Définition 2.14** (TAGED positif)
*Un TAGED positif* $A$ *est un 5-uplet* $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta, E)$ *où :*
- $(\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ *est automate d'arbres,*
- $E \subseteq \mathcal{Q} \times \mathcal{Q}$ *est une relation réflexive et symétrique.*

On notera $ta(A)$ l'automate d'arbres $(\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ correspondant au TAGED positif $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta, E)$.

Une exécution réussie d'un terme $t \in \mathcal{T}(\mathcal{F})$ pour un TAGED positif $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta, E)$, est une exécution réussie $\rho$, de $t$ pour $ta(A)$, qui satisfait la condition suivante : pour toutes positions $p_1$, $p_2 \in \mathcal{P}os(t)$, si $(\rho(p_1), \rho(p_2)) \in E$ alors $t|_{p_1} = t|_{p_2}$.

Le langage reconnu par un TAGED positif $A$, noté $L(A)$, est l'ensemble des termes $t$ de $\mathcal{T}(\mathcal{F})$ pour lesquels il existe une exécution réussie de $A$.

D'après cette définition d'une exécution pour les TAGED positifs, on peut en déduire que pour tout TAGED positif $A$, on a :

$$L(A) \subseteq L(ta(A)).$$

♢ **Exemple 2.15**
*Soit $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta, E)$ un TAGED positif où :*
- $\mathcal{F} = \{g^2, f^1, a^0\}$
- $\mathcal{Q} = \{q_a, q_=, q_g\}$
- $\mathcal{Q}_f = \{q_g\}$
- $\Delta = \{a \to q_a, a \to q_=, f(q_a) \to q_=, f(q_a) \to q_a, g(q_=, q_=) \to q_g\}$
- $E = \{(q_=, q_=)\}$

*Cet automate reconnaît le langage suivant :*

$$L(A) = \{g(f^n(a), f^n(a)) \mid n \in \mathbb{N}\},$$

*qui n'est pas un langage régulier. De plus, le langage reconnu par l'automate d'arbres $ta(A)$ est le suivant :*

$$L(ta(A)) = \{g(f^m(a), f^n(a)) \mid m, n \in \mathbb{N}\}.$$

*On a $L(A) \subseteq L(ta(A))$ dans ce cas là.*

Pour les TAGED positifs, le problème de vacuité se résout en temps exponentiel [FTT08, Theorem 1]. Les problèmes de l'universalité et de l'inclusion sont indécidables [FTT08, Proposition 5].

De plus, le problème de vacuité pour les TAGED [FTT08, Theorem 1] est résolu (positivement) dans [BCG+10], où les auteurs proposent un algorithme de décision du vide pour les TAGC (Tree Automata with Global Constraints), classe d'automates généralisant les TAGED.

## 2.4 La réécriture par complétion

Dans cette section, nous allons décrire formellement le mécanisme de la complétion d'automates d'arbres, précédemment décrit dans [Gen98, FGT04], permettant de calculer une sur-approximation des termes atteignables par réécriture. A partir d'un système de réécriture $R$ et d'un ensemble de termes $E$, la procédure de complétion d'automate d'arbres permet de calculer une sur-approximation de l'ensemble $R^*(E)$, étant donné que ce dernier n'est généralement pas calculable.

La procédure de complétion d'automates d'arbres est au centre de cette thèse.

Soit $R$ un système de réécriture linéaire à gauche et $A_0$ un automate d'arbres. La procédure de complétion d'automate d'arbres calcule un automate $A_k$ tel que $R^*(L(A_0)) \subseteq L(A_k)$. Pour obtenir l'automate $A_k$ à partir de $A_0$, une suite d'automates $A_0$, $A_1$, $A_2$, …, $A_k$ est calculée, où un automate d'arbres $A_{i+1}$ est calculé à partir de l'automate d'arbres $A_i$ $(0 \leq i < k)$. La complétion s'arrête quand on obtient un automate $A_{k+1}$ tel que $A_{k+1} = A_k$. Dans ce cas, on dit de $A_k$ qu'il est un point fixe.

Pour calculer un automate d'arbres $_{i+1}$ à partir de $_{i}$, une étape de complétion est réalisée où l'on recherche toutes les *paires critiques* existantes entre $_{R}$ et $_{A_i}$. Une paire critique pour $_{i}$ est un triplet ( ), où $: X$ est une substitution et une règle de réécriture, et telle que $_{A_i}$ et $_{A_i}$ . Pour chaque paire critique ( ) trouvée entre $_{R}$ et $_{A_i}$, la règle est ajoutée aux transitions de $_{i}$, afin de construire $_{i+1}$ (figure 2.4).

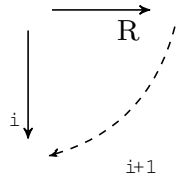



FIGURE 2.4 – Paire critique

Cependant, les nouvelles règles ajoutées ne sont pas forcément de la forme f( $_{1}$ ::: $_{n}$) , où f est un symbole et $_{1}$, :::, $_{n}$, des états. On doit donc réaliser une étape de normalisation des nouvelles règles. La normalisation d'une règle consiste en un "éclatement" d'une règle de façon à obtenir un ensemble de transitions normalisées $\Delta_n$ tel que $_{n}$ . Par exemple, considérons la règle f (g(a) ) . Une normalisation possible donnerait l'ensemble $\Delta_n$ suivant : $\Delta_n = f$a $_{a}$ $_{b}$ g( $_{a}$) $_{g}$ f ( $_{g}$ $_{b}$) *g*. On a bien f (g(a) ) $_{n}$ . On remarque qu'il faut utiliser des états afin de normaliser une règle. Dans l'exemple, on a dû utiliser trois états $_{a}$, $_{b}$, $_{g}$ pour normaliser cette règle. Ces états peuvent être choisi arbitrairement et c'est ce choix qui va influer sur le futur déroulement de la complétion et sur l'approximation. Si on reprend l'exemple et que l'on se limite cette fois à utiliser deux états pour normaliser la transition f (g(a) ) , la normalisation peut donner l'ensemble $\Delta_n$ suivant : $\Delta_n = f$a $_{a}$ $_{b}$ g( $_{a}$) $_{a}$ f ( $_{a}$ $_{b}$) *g*. On a toujours f (g(a) ) $_{n}$ , et on a aussi f (g (a) ) $_{n}$ .

***Complétion d'automates d'arbres.*** Dans cette section on notera un alphabet fini, $X$ un ensemble fini de variables, un ensemble fini d'états et R un système de réécriture.

Nous allons tout d'abord définir une fonction d'approximation. Celle-ci permet d'associer à une position un état.

DØ nition 2.16 (fonction d'approximation)
*Une fonction d'approximation* *est une fonction* : ((R (*X* ) ) N ) *telle que* ( )( ) = .

Pour une fonction d'approximation quelconque on notera m( ) l'image de .

Maintenant nous allons définir formellement une fonction de normalisation qui permettra de normaliser une transition issue d'une paire critique à l'aide d'une fonction d'approximation.

DØ nition 2.17 ( normalisation)
*Soient* *une fonction d'approximation,* $\Delta$ *un ensemble de transitions,* R *avec* $V$a ( ) $\theta$ $V$a (

A partir des deux définitions précédentes, nous allons définir les modifications faites (ajouts de transitions et d'états à partir de paires critiques) à un automate d'arbres $A_i$ lors d'une étape de complétion. On notera $C_\gamma^R(A_i)$ l'automate obtenu après une étape de complétion d'un automate d'arbres $A_i$ par un système de réécriture $R$ et une fonction d'approximation $\gamma$.

△ **Définition 2.18** (Complétion d'automate)
*Soit $A = (\mathcal{F}, \mathcal{Q}_A, \mathcal{Q}_f, \Delta)$ un automate d'arbres (avec $\mathcal{Q}_A \subseteq \mathcal{Q}$) et $\gamma$ une fonction d'approximation.*
*On pose $C_\gamma^R(A) = (\mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta')$ l'automate d'arbres défini par :*

$$\Delta' = \Delta \cup \bigcup_{l \to r \in R,\ Var(r) \subseteq Var(l),\ \sigma : \mathcal{X} \to \mathcal{Q},\ l\sigma \to_A^* q} Norm_\gamma(l \to r, \sigma, q)$$

$$\mathcal{Q}' = \{q \mid c \to q \in \Delta'\}$$

Notons que $\mathcal{Q}' \subseteq \mathcal{Q}$ et que l'ensemble des états finaux est inchangé.

Il s'en suit une proposition énonçant la relation entre l'ensemble des termes atteignables calculé après une étape de réécriture et l'ensemble des termes atteignables après une étape de complétion.

Dans [FGT04] il est défini une *condition de linéarité à gauche* pour un automate d'arbres et un système de réécriture. Cette condition assure une forme déterminisme pour un sous-ensemble des états de l'automate d'arbres qui substitueront les variables apparaissant plusieurs fois dans les parties gauches des règles de réécriture. Avec cette condition, le théorème 1 de [FGT04] assure la correction de la procédure de completion vis-à-vis du système de réécriture. C'est-à-dire que pour un système de réécriture $R$ et un automate d'arbres $A$ satisfaisant la condition de linéarité à gauche, que pour toute fonction d'approximation $\gamma$ on a :

$$R^*(L(A)) \subseteq L((C_\gamma^R)^{(N)}(A))$$

si $(C_\gamma^R)^{(N)}(A) = (C_\gamma^R)^{(N+1)}(A)$ (où $N$ est un entier strictement positif).

La proposition et le théorème qui suivent sont des adaptations de ce théorème.

♦ **Proposition 2.19** (Conséquence de [FGT04, Théorème 1])
*Soit $A$ un automate d'arbres et $R$ système de réécriture tel que $A$ est déterministe ou $R$ est linéaire à gauche, et, de plus, pour chaque $l \to r \in R$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. On a, pour toute fonction d'approximation $\gamma$ :*

$$L(A) \cup R(L(A)) \subseteq C_\gamma^R(A).$$

Cette proposition s'étend après $N$ étapes de complétion.

★ **Théorème 2.20** (Conséquence de [FGT04, Théorème 1])
*Soient $A$ et $R$ respectivement un automate d'arbres et un système de réécriture linéaire à gauche tel que pour chaque $l \to r \in R$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Pour chaque fonction d'approximation, s'il existe $N \in \mathbb{N}$ tel que $(C_\gamma^R)^{(N)}(A) = (C_\gamma^R)^{(N+1)}(A)$, alors*

$$R^*(L(A)) \subseteq L((C_\gamma^R)^{(N)}(A)).$$

Nous illustrons la réalisation des étapes de complétion et le calcul d'un automate point fixe dans l'exemple suivant.

♢ **Exemple 2.21**
*Dans cet exemple, on utilisera les automates de mots pour illustrer la procédure de complétion. On utilisera une notation fonctionnelle pour représenter les mots reconnus par les automates de*

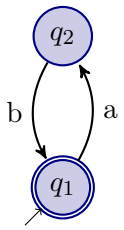



FIGURE 2.5 – Automate initial $A_0$

*mot. Par exemple, le mot abab est représenté par le terme $b(a(b(a(\omega))))$, où $\omega$ est un symbole d'arité 0.*

*Soit $A_0$ un automate reconnaissant les termes du type $b(a(\dots b(a(\omega))\dots))$ (figure 2.5). Soit le système de réécriture $R = \{b(a(x)) \to a(b(x))\}$.*

***Première étape de complétion :***
*On a la paire critique $(b(a(x)) \to a(b(x)), \{x \to q_1\}, q_1)$ et on choisit une fonction d'approximation telle que :*

$$\gamma_1(b(a(x)) \to a(b(x)), \{x \to q_1\}, q_1) = \{1 \to q_3\}.$$

*La règle $a(b(q_1)) \to q_1$ doit être ajoutée aux transitions de l'automate $A_0$, mais elle doit tout d'abord être normalisée. Ce qui donne les transitions $Norm_{\gamma_1}(b(a(x)) \to a(b(x)), \{x \to q_1\}, q_1) = \{a(q_3) \to q_1, b(q_1) \to q_3\}$. On obtient l'automate $A_1$ $(= C^R_{\gamma_1}(A_0))$ représenté par la figure 2.6.*

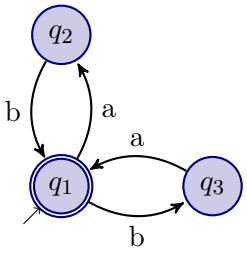



FIGURE 2.6 – Automate $A_1$

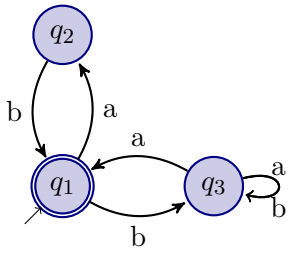



FIGURE 2.7 – Automate $A_2$

***Deuxième étape de complétion :***
*Avec l'automate $A_1$, on a une nouvelle paire critique $(b(a(x)) \to a(b(x)), \{x \to q_3\}, q_3)$ et on choisit la fonction d'approximation telle que :*

$$\gamma_1(b(a(x)) \to a(b(x)), \{x \to q_3\}, q_3) = \{1 \to q_3\}.$$

*Les transitions $Norm_{\gamma_1}(b(a(x)) \to a(b(x)), \{x \to q_3\}, q_3) = \{a(q_3) \to q_3, b(q_3) \to q_3\}$ sont ajoutées aux transitions de l'automate $A_1$. On obtient l'automate point fixe $A_2$ $(= C^R_{\gamma_1}(A_1))$ de la figure 2.7.*

*L'automate $A_2$ est un point fixe car il n'y a plus de paire critique.*

***Complétion exacte.*** Maintenant, nous allons nous intéresser à la construction d'une fonction d'approximation particulière dite $(A, R)-$exacte. Cette fonction d'approximation va nous permettre de construire un automate dit exact issu d'une étape de complétion de l'automate $A$ à l'aide du système de réécriture $R$.

Cette classe de fonctions d'approximation permet de calculer uniquement des termes accessibles lors de la complétion.

△ **Définition 2.22** (Fonction d'approximation $(A,R)-$exacte)
*Soit $A=(\mathcal{F},\mathcal{Q}',\mathcal{Q}'_f,\Delta')$ un automate d'arbres, $R$ un système de réécriture. Une fonction d'approximation $\gamma$ est dite $(A,R)-$exacte si :*

$$(1)\quad \gamma \text{ est injective}$$
$$(2)\quad Im(\gamma)\cap\mathcal{Q}'=\emptyset$$

Lors de la normalisation d'une transition avec une fonction d'approximation $(A,R)-$exacte, seulement des nouveaux états sont utilisés, ce qui permet de calculer uniquement des termes accessibles.

♦ **Proposition 2.23** (Propriété d'une fonction d'approximation $(A,R)-$exacte)
*Soit $A=(\mathcal{F},\mathcal{Q},\mathcal{Q}_f,\Delta)$ un automate d'arbres et $R$ un système de réécriture tel pour chaque $l\to r\in R$, $\mathcal{V}ar(r)\subseteq\mathcal{V}ar(l)$. Soient $\gamma_1$ et $\gamma_2$ deux fonctions d'approximation. Si elles sont $(A,R)-$exactes alors $C^R_{\gamma_1}(A)$ et $C^R_{\gamma_2}(A)$ sont égaux au renommage des états près.*

*Démonstration.* Pour toute règle de réécriture $l\to r$ de $R$, pour toute substitution $\sigma$, pour tout état $q$ de $\mathcal{Q}$ et pour toute position $p$ de $\mathcal{P}os_{\mathcal{F}}(r)$, on renomme $\gamma_1(l\to r,\sigma,q)(p)$ par $\gamma_2(l\to r,\sigma,q)(p)$. □

△ **Définition 2.24** (Automate exact)
*Soit $A$ un automate d'arbres, soit $R$ un système de réécriture et $\alpha$ une fonction d'approximation $(A,R)-$exacte. L'automate d'arbres $A^e=C^R_\alpha(A)$ est appelé automate exact de $A$.*

La réalisation d'étapes de complétion à l'aide d'une fonction d'approximation $(A,R)-$exacte est illustrée dans l'exemple suivant.

♢ **Exemple 2.25**
*Dans cet exemple nous allons illustrer la complétion d'automate d'arbres en utilisant des fonctions d'approximation exactes. On reprend le même automate $A_0$ et le même système de réécriture $R$ que pour l'exemple 2.21.*

***Première étape de complétion :***
*Tout comme la première étape de complétion de l'exemple 2.21, on a la paire critique $(b(a(x))\to a(b(x)),\{x\to q_1\},q_1)$ et on choisit la même fonction d'approximation $\gamma_1$ (qui est une fonction d'approximation $(A_0,R)-$exacte). On obtient l'automate $A_1$.*

***Deuxième étape de complétion :***
*On a la paire critique $(b(a(x))\to a(b(x)),\{x\to q_3\},q_3)$, et on prend la fonction d'approximation $(A_1,R)-$exacte suivante :*

$$\alpha_1(b(a(x))\to a(b(x)),\{x\to q_3\},q_3)=\{1\to q_4\}.$$

*On rajoute les transitions $Norm_{\alpha_1}(b(a(x))\to a(b(x)),\{x\to q_3\},q_3)=\{a(q_4)\to q_3,b(q_3)\to q_4\}$ aux transitions de l'automate $A_1$. On obtient l'automate $A_2^e$ représenté par la figure 2.8. L'automate $A_2^e$ est l'automate exact de $A_2$.*

*On remarque que les prochaines étapes de complétion vont toutes produire un nouvel état et deux nouvelles transitions, si on utilise une fonction d'approximation $(A,R)-$exacte. En effet à chaque fois le nouvel état reconnaît un terme de la forme $b(a(...))$ (le mot $ab$), ce qui implique l'ajout de nouvelles transitions et d'un nouvel état. Dans ce cas, on remarque que la complétion ne termine pas si on utilise une fonction d'approximation $(A,R)-$exacte.*

♦ **Proposition 2.26** (Propriété d'un automate exact)
*Soit $A$ un automate d'arbres, $R$ un système de réécriture linéaire tel que $Var(r)\subseteq Var(l)$ pour chaque règle de $R$ et $\gamma$ une fonction d'approximation $(A,R)-$exacte, alors :*

$$\mathcal{L}(A)\cup R(\mathcal{L}(A))\subseteq C^R_\gamma(A)\subseteq R^*(\mathcal{L}(A))$$

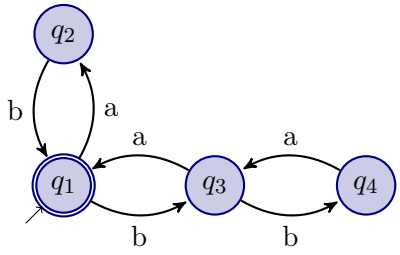



FIGURE 2.8 – Automate $A_2^e$

Dans ce chapitre nous avons introduit toutes les notions indispensables à la compréhension de la grande majeure partie de cette thèse. Notons pour finir qu'il est présenté dans [GR10] une technique avancée de complétion utilisable avec des théories equationnelles.

Nous allons maintenant présenter deux applications de la complétion d'automates d'arbres : la vérification de spécifications CCS et la semi-décision de deux problèmes indécidables pour les machines de Turing.

Deuxième partie

# Analyse d'accessibilité par réécriture

# 3

# Vérification de spécifications CCS

## Sommaire



Dans ce chapitre, il sera question de vérifier des propriétés d'accessibilité pour des systèmes à états infinis spécifiés en CCS. On se limitera à une variante de CCS sans renommage, pour laquelle le problème d'atteignabilité est indécidable [BGZ03]. Le but est de montrer que la réécriture d'approximation s'applique efficacement à la vérification de spécifications CCS.

Il existe plusieurs outils permettant de vérifier des programmes CCS, comme le Edinburgh Concurrency Workbench [RJB94], le Concurrency Workbench North Carolina [CS96] et XMC [RRR+97], qui s'intéressent à la vérification de systèmes à états finis, tandis que notre méthode s'adresse aux systèmes à états infinis.

Dans [FFGI94], les auteurs définissent une procédure de semi-décision permettant la vérification des propriétés ACTL [DNV90] (propriétés fondées sur les actions) pour les systèmes à états infinis. La méthode présentée dans cette section ne permet pas de vérifier des propriétés CTL, mais permet de vérifier des propriétés d'atteignabilité s'appuyant sur les actions et sur les expressions CCS. Ce type de propriétés d'atteignabilité est exprimé sous la forme d'un automate d'arbres, plutôt qu'à l'aide d'une logique temporelle.

Plus précisément, le programme CCS analysé et la sémantique de CCS sont transformés en un système de réécriture et en un automate d'arbre. Sachant que la sémantique opérationnelle du programme CCS s'exprime en termes de système de transitions étiquetées, la technique de complétion définie dans la section 2.4 nous permet de calculer une sur-approximation :

– des états atteignables,
– des traces permettant d'atteindre ces états.

Ensuite l'intersection entre cette sur-approximation et un ensemble régulier de termes, représentant des états et des traces, permet de savoir si ces derniers sont atteignables. Illustrée par la figure 3, cette approche sera testée sur le protocole ABP et des composants électroniques. De plus, ces exemples permettront d'illustrer le problème de la définition de la fonction d'approximation évoqué à la section 2.4.

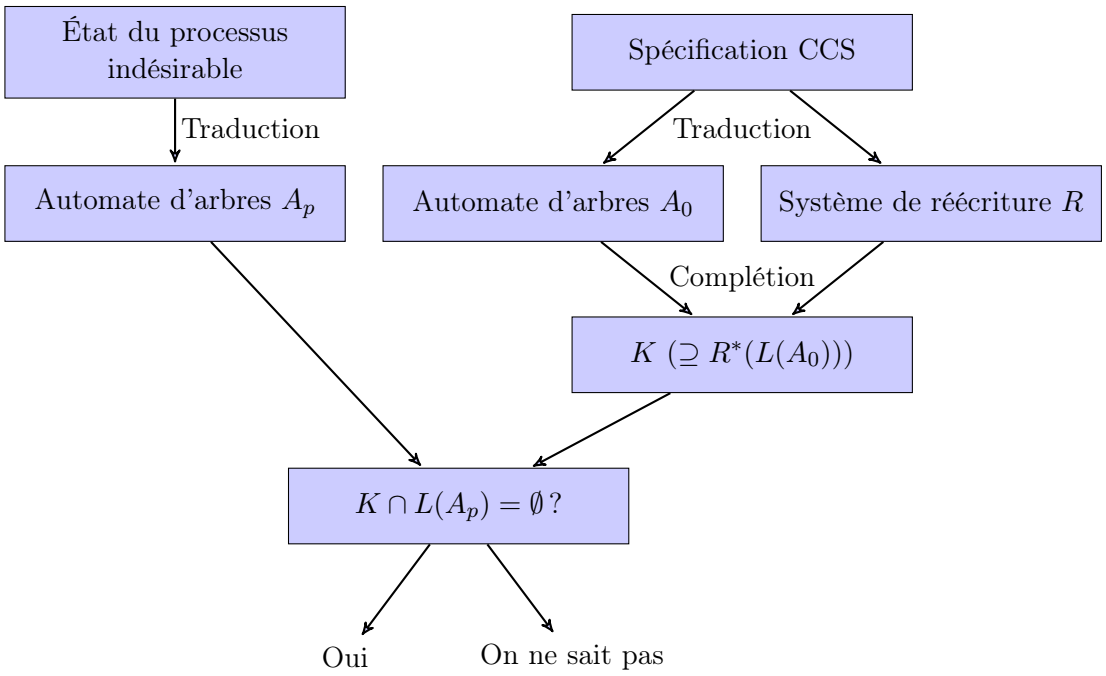



FIGURE 3.1 – Procédure pour l'analyse de programmes CCS par réécriture d'approximation

Dans la section 3.1, nous allons tout d'abord présenter l'algèbre de processus CCS sans le renommage. Ensuite, dans la section 3.2, nous verrons comment traduire un programme CCS en un système de réécriture et en un automate d'arbres. Puis quelques expérimentations seront présentées dans les sections 3.3 et 3.4.

## 3.1 L'algèbre de processus CCS

Le *Calculus of Communicating Systems* ou (CCS) est un algèbre de processus introduit par Milner [Mil89], mettant l'accent sur la communication entre des systèmes. Il existe beaucoup d'autres algèbres de processus comme CSP, LOTOS, ou encore SDL.

### 3.1.1 Syntaxe

Soit $\mathcal{A} = \{a, b, c, \ldots\}$ un ensemble infini appelé ensemble de canaux. Soit $\bar{\mathcal{A}} = \{\bar{a}, \bar{b}, \bar{c}, \ldots\}$ un ensemble infini, copie disjointe de $\mathcal{A}$, appelé ensemble de co-canaux. On pose $\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$ l'ensemble des labels. Soit $\tau \notin \mathcal{L}$, un élément appelé action silencieuse. On notera $Act = \mathcal{L} \cup \{\tau\}$ l'ensemble des actions. Soit $\mathcal{P}$ un ensemble disjoint de $Act$ appelé ensemble des constantes. On suppose que $\mathcal{P}$ contient un élément particulier, noté $\mathbf{0}$, et appelé le processus inactif.

△ **Définition 3.1**
*Les expressions CCS $E$, $E_1$ et $E_2$ sont définies par la grammaire suivante :*

$$E, E_1, E_2 := \alpha.E \mid E_1 + E_2 \mid E_1 \parallel E_2 \mid E \setminus \ell \mid \mathbf{0} \mid P$$

*où $\alpha \in Act$, $\ell \in \mathcal{L}$ et $P \in \mathcal{P}$.*

On notera $\mathcal{E}$ l'ensemble des expressions CCS construites d'après la définition 3.1. Notons que pour enlever toute ambiguïté, les expressions CCS sont souvent écrites avec des parenthèses.

♢ **Exemple 3.2**
*Soient $a, \bar{b}, b, c$ des actions. Soient $R$ et $Q$ des constantes. Les expressions $(a.\bar{b}.\mathbf{0}) + (c.R \parallel a.Q)$ et $(a.b.Q \parallel \bar{b}.Q) \setminus b$ sont des expressions CCS. Les expressions $a.R.b$ et $a.R \parallel R.Q$ ne sont pas des expressions CCS.*

Une équation CCS est un couple $(P, E)$ où $P \in \mathcal{P}$ et $E \in \mathcal{E}$. Une équation CCS peut aussi être notée $P \stackrel{def}{=} E$.

♢ **Exemple 3.3**
*En reprenant les notations de l'exemple 3.2, $Q \stackrel{def}{=} (a.b.Q \parallel \bar{b}.Q) \setminus b$ est une équation CCS.*

On note $Action(E)$ l'ensemble des actions utilisées pour définir une expression $E$. Il est défini par induction de la manière suivante :

$Action(\alpha.E) = \{\alpha\} \cup Action(E)$
$Action(E \setminus \ell) = Action(E)$
$Action(\mathbf{0}) = \emptyset$
$Action(P) = \emptyset$ (avec $P \in \mathcal{P}$)
$Action(E_1 + E_2) = Action(E_1 \parallel E_2) = Action(E_1) \cup Action(E_2)$

On définit aussi l'ensemble des actions utilisées pour une équation $P \stackrel{def}{=} E$, par

$$Action(P \stackrel{def}{=} E) = Action(E).$$

On note $ResAction(E)$ l'ensemble des actions participants aux opérations de restriction $(E_1 \setminus \ell)$ compris dans $E$. Cet ensemble est défini par induction de la façon suivante pour une expression $E$ :

$ResAction(E \setminus \ell) = ResAction(E) \cup \{\ell\}$
$ResAction(\alpha.E) = ResAction(E)$
$ResAction(\mathbf{0}) = \emptyset$
$ResAction(E_1 + E_2) = ResAction(E_1 \parallel E_2) = ResAction(E_1) \cup ResAction(E_2)$

On définit aussi l'ensemble des restrictions pour une équation $P \stackrel{def}{=} E$, par

$$ResAction(P \stackrel{def}{=} E) = ResAction(E).$$

Pour toute expression $E$, on note $Subterm(E)$ un ensemble d'expressions CCS défini par induction :

$Subterm(\alpha.E) = \{\alpha.E\} \cup Subterm(E)$

$Subterm(E \setminus \ell) = \{E \setminus \ell\} \cup Subterm(E)$
$Subterm(\mathbf{0}) = \emptyset$
$Subterm(E_1 + E_2) = \{E_1 + E_2\} \cup Subterm(E_1) \cup Subterm(E_2)$
$Subterm(E_1 \parallel E_2) = \{E_1 \parallel E_2\} \cup Subterm(E_1) \cup Subterm(E_2)$
On dit qu'une expression CCS $E'$ est un sous-terme de $E$, ou $E$ contient $E'$, si $E' \in Subterm(E)$.

♢ **Exemple 3.4**
*En reprenant l'exemple 3.2 on a :*
$Action((a.b.Q \parallel \bar{b}.Q) \setminus b) = \{a, b, \bar{b}\}$
$ResAction((a.b.Q \parallel \bar{b}.Q) \setminus b) = \{b\}$
$Subterm((a.b.Q \parallel \bar{b}.Q) \setminus b) = \{((a.b.Q \parallel \bar{b}.Q) \setminus b), (a.b.Q \parallel \bar{b}.Q), a.b.Q, b.Q, \bar{b}.Q, Q\}$

### 3.1.2 Programme CCS

Un programme CCS $X$ est un triplet $X = (\Lambda, \Gamma, P_0)$ où $\Lambda \subseteq Act$, $\Gamma \subseteq \mathcal{P} \times \mathcal{E}$ est un ensemble fini d'équations n'utilisant que des lettres de $\Lambda$, et $P_0 \in \{P \mid (P, E) \in \Gamma\}$ est le nom du processus de tête, qui définit le système complet habituellement.

♢ **Exemple 3.5**
$X = (\{a, \bar{b}, c\}, \{(R, a.\bar{b}.P + \bar{b}.a.P), (P, c.R)\}, R)$ *est un programme CCS.*

### 3.1.3 Sémantique

La sémantique opérationnelle d'un programme CCS $X = (\Lambda, \Gamma, P_0)$ est définie par un système de transitions étiquetées (STE) avec des expressions CCS dans $\mathcal{E}$ pour états et une relation de transition $T_{CCS} \subseteq \mathcal{E} \times \Lambda \times \mathcal{E}$, qui obéit à des règles d'inférence de la figure 3.2. Une transition $E \xrightarrow{\alpha} E'$ peut aussi être notée $(E, \alpha, E') \in T_E$. Dans ce contexte, les expressions CCS $E$ et $E'$ sont appelées états. Un système de transitions étiquetées est un graphe dont les arêtes sont étiquetées par des actions dans $\Lambda$. Nous ne le définirons pas plus précisément car cela n'est pas utile dans cette thèse.

$$\textbf{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \textbf{Com}_\textbf{1}\ \frac{E \xrightarrow{\alpha} E'}{E \parallel F \xrightarrow{\alpha} E' \parallel F}$$

$$\textbf{Sum}_\textbf{1}\ \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 + E_2 \xrightarrow{\alpha} E_1'} \qquad \textbf{Com}_\textbf{2}\ \frac{F \xrightarrow{\alpha} F'}{E \parallel F \xrightarrow{\alpha} E \parallel F'}$$

$$\textbf{Sum}_\textbf{2}\ \frac{E_2 \xrightarrow{\alpha} E_2'}{E_1 + E_2 \xrightarrow{\alpha} E_2'} \qquad \textbf{Com}_\textbf{3}\ \frac{E \xrightarrow{a} E' \quad F \xrightarrow{\bar{a}} F'}{E \parallel F \xrightarrow{\tau} E' \parallel F'}$$

$$\textbf{Res}\ \frac{E \xrightarrow{\alpha} E'}{E \setminus \ell \xrightarrow{\alpha} E' \setminus \ell}\ \text{si } \alpha, \bar{\alpha} \neq \ell$$

$$\textbf{Con}\ \frac{E \xrightarrow{\alpha} E'}{P \xrightarrow{\alpha} E'}\ \text{si } (P, E) \in \Gamma$$

FIGURE 3.2 – Règles d'inférence pour CCS

Une expression CCS $E$ peut réaliser une action $\alpha$ et devenir l'expression CCS $E'$ si la transition $E \xrightarrow{\alpha} E'$ peut être inférée par les règles de la figure 3.2.

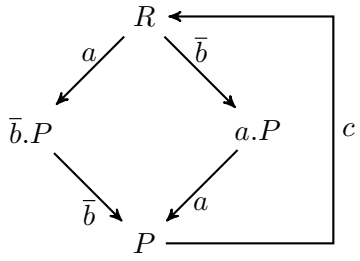



FIGURE 3.3 – STE associé au programme CCS $X$

♢ **Exemple 3.6**
*Soit $X$ le programme CCS de l'exemple 3.5. La transition $R \xrightarrow{a} \bar{b}.P$, peut être inférée par l'application des règles* **Con**, $\mathbf{Sum_1}$ *et* **Act**. *Le STE associé au programme CCS $X$ se trouve dans la figure 3.3.*

### 3.1.4 Dérivatifs

Si on a $E \xrightarrow{\alpha} E'$, le couple $(\alpha, E')$ est appelé *dérivatif immédiat* de $E$.

Si on a $E \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_n} E'$ avec $\alpha_0, \ldots, \alpha_n$ une séquence d'actions, le couple $(\alpha_0 \ldots \alpha_n, E')$ est appelé *dérivatif* de $E$. On note $deriv(E)$ l'ensemble des dérivatifs de $E$.

Soit $X = (\Lambda, \Gamma, P_0)$ un programme CCS. L'ensemble de dérivatifs atteignables par $X$ est l'ensemble des dérivatifs de $P_0$ soit $deriv(P_0)$.

♢ **Exemple 3.7**
*En reprenant l'équation de l'exemple 3.3, on a :*

$(a, (b.Q \parallel \bar{b}.Q) \setminus b)$ *est un dérivatif immédiat de $Q$.*

$(a\tau a\tau a\tau, Q)$ *est un dérivatif de $Q$.*

$\{(a(\tau a)^*, (b.Q \parallel \bar{b}.Q) \setminus b), ((a\tau)^*, Q)\}$ *est l'ensemble $deriv(Q)$ des dérivatifs de $Q$.*

Nous allons maintenant nous intéresser à la modélisation sous la forme d'un système de réécriture et d'un automate d'arbres d'une spécification CCS et des règles d'inférence pour CCS. L'objectif est d'utiliser la complétion d'automates d'arbres pour calculer un sur-ensemble des dérivatifs atteignables par la spécification.

## 3.2 Système de réécriture et CCS

Dans cette section nous allons expliquer comment transformer un programme CCS et la sémantique de CCS en un système de réécriture $R$ et un automate d'arbres $A$ pour les configurations initiales. Grâce à cette transformation et à la technique expliquée dans la section 2.4, on pourra calculer une sur-approximation de $R^*(L(A))$, représentant une sur-approximation des dérivatifs atteignables par un programme CCS.

Dans un premier temps, nous allons voir dans les sections 3.2.1 et 3.2.2 comment encoder une expression CCS et un dérivatif. Puis nous allons voir comment la sémantique et les équations (d'un programme CCS) sont transformées en un système de réécriture dans les sections 3.2.3 et 3.2.4. Et finalement, nous montrerons comment vérifier des propriétés d'une spécification CCS dans la section 3.2.5.

### 3.2.1 Codage d'une expression CCS

Soit $\mathcal{F}_{CCS}$ un alphabet tel que $\mathcal{F}_{CCS} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3$, où $\mathcal{F}_0 = \{\mathbf{0}\} \cup \mathcal{P} \cup Act$, $\mathcal{F}_1 = \{bar\}$, $\mathcal{F}_2 = \{Pre, Sum, Com, Res, Sys\}$. Une expression CCS $E$ est codée en un terme de $\mathcal{T}(\mathcal{F}_{CCS})$ par induction sur sa structure, à l'aide de la fonction $\Phi : \mathcal{E} \to \mathcal{T}(\mathcal{F}_{CCS})$. Cette fonction est définie de la manière suivante :

$\Phi(\alpha.E) = Pre(\Phi(\alpha), \Phi(E))$

$\Phi(E_1 + E_2) = Sum(\Phi(E_1), \Phi(E_2))$

$\Phi(E_1 \parallel E_2) = Com(\Phi(E_1), \Phi(E_2))$

$\Phi(P) = P$, si $P \in \mathcal{P}$

$\Phi(E \setminus \ell) = Res(\Phi(E), \Phi(\ell))$

$\Phi(\mathbf{0}) = \mathbf{0}$

$$\Phi(\alpha) = \begin{cases} \alpha & \text{si } \alpha \in \mathcal{A} \cup \{\tau\} \\ bar(a) & \text{si } \alpha = \bar{a} \text{ et } \alpha \in \bar{\mathcal{A}} \end{cases}$$

♢ **Exemple 3.8**
*Soit* $E = (a.b.\mathbf{0} + b.\mathbf{0}) \parallel c.\mathbf{0}$ *une expression CCS. Le terme correspondant à* $E$ *est :*

$$\begin{aligned} \Phi(E) &= Com(\phi(a.b.\mathbf{0} + b.\mathbf{0}), \phi(c.\mathbf{0}) \\ &= Com(Sum(\phi(a.b.\mathbf{0}), \phi(b.\mathbf{0})), Pre(c, \phi(\mathbf{0}))) \\ &= Com(Sum(Pre(a, \phi(b.\mathbf{0})), Pre(b, \phi(\mathbf{0}))), Pre(c, \mathbf{0})) \\ &= Com(Sum(Pre(a, Pre(b, \phi(\mathbf{0}))), Pre(b, \mathbf{0})), Pre(c, \mathbf{0})) \\ &= Com(Sum(Pre(a, Pre(b, \mathbf{0})), Pre(b, \mathbf{0})), Pre(c, \mathbf{0})) \end{aligned}$$

### 3.2.2 Codage d'un dérivatif

Soit $E$ une expression CCS. Nous proposons d'encoder un dérivatif $(\alpha_0 \dots \alpha_n, E)$ en un terme de la forme $Sys(\alpha_0, Sys(\dots, Sys(\alpha_n, \Phi(E))))$ où $Sys$ est un symbole d'arité 2, et la fonction $\Phi(E)$ définie dans la section 3.2.1. Formellement, on pose la fonction $\Psi : deriv(E) \to \mathcal{T}(\mathcal{F}_{CCS})$ définie par :

$\Psi((\alpha, E)) = Sys(\Phi(\alpha), \Phi(E))$, si $\alpha \in Act$, et

$\Psi((\alpha_0 \dots \alpha_n, E)) = Sys(\Phi(\alpha_0), \Psi((\alpha_1 \dots \alpha_n, E)))$ si les $\alpha_i$ sont dans $Act$ et $n > 0$.

Le symbole $Sys$ sera utilisé dans la section suivante pour la définition du système de réécriture. De plus l'encodage des dérivatifs sous forme de termes servira pour la modélisation des propriétés à vérifier.

### 3.2.3 Transformation de la sémantique de CCS en système de réécriture

Dans cette section nous allons voir comment obtenir un système de réécriture permettant d'obtenir les dérivatifs d'une expression CCS. Ce système de réécriture s'inspire des règles d'inférence de CCS.

Intuitivement, nous allons voir comment les règles de la figure 3.4 sont construites en prenant l'exemple de la règle $\rho_2$.

Nous partons de la règle d'inférence $\mathbf{Sum_1}$. La règle $\mathbf{Sum_1}$ peut se lire : "$(\alpha, E'_1)$ est un dérivatif de $E_1 + E_2$ si $(\alpha, E'_1)$ est un dérivatif de $E_1$". Dans un premier temps on peut en déduire la règle de réécriture $Sum(E_1, E_2) \to Sys(\alpha, E'_1)$, puis dans un deuxième temps on a $Sum(Sys(\alpha, E'_1), E_2) \to Sys(\alpha, E'_1)$ car on a par hypothèse que $(\alpha, E'_1)$ est un dérivatif de $E_1$.

Si on considère $\alpha$, $E_1'$ et $E_2$ comme des variables on obtient bien la règle de réécriture $\rho_2$ du tableau 3.4. Pour résumer, la règle d'inférence

$$\mathbf{Sum_1}\ \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 + E_2 \xrightarrow{\alpha} E_1'}$$

se transforme en la règle de réécriture

$$Sum(Sys(x, y), z) \to Sys(x, y).$$

C'est sur ce même principe que les autres règles de réécriture ont été construites.

| | | | |
|---|---|---|---|
| $\rho_1$ | $Pre(x, y)$ | $\to$ | $Sys(x, y)$ |
| $\rho_2$ | $Sum(Sys(x, y), z)$ | $\to$ | $Sys(x, y)$ |
| $\rho_3$ | $Sum(z, Sys(x, y))$ | $\to$ | $Sys(x, y)$ |
| $\rho_4$ | $Com(Sys(x, y), z)$ | $\to$ | $Sys(x, Com(y, z))$ |
| $\rho_5$ | $Com(z, Sys(x, y))$ | $\to$ | $Sys(x, Com(z, y))$ |
| $\rho_6$ | $Com(Sys(x, y), Sys(bar(x), z))$ | $\to$ | $Sys(\tau, Com(y, z))$ |
| $\rho_7$ | $Com(Sys(bar(x), y), Sys(x, z))$ | $\to$ | $Sys(\tau, Com(y, z))$ |
| $\rho_8$ | $Res(Sys(x, y), z)$ | $\to$ | $Sys(x, Res(y, z))$ |

FIGURE 3.4 – Règles de réécriture pour la sémantique de CCS

Soit le système de réécriture linéaire à droite $R_{CCS} = \{\rho_1,\ \rho_2,\ \rho_3,\ \rho_4,\ \rho_5,\ \rho_6,\ \rho_7,\ \rho_8\}$ avec les règles dans la figure 3.4. En analysant la forme de ce système de réécriture, on voit qu'il ne convient pas (pour notre approche) pour deux raisons :

- Étant donné que l'objectif est d'utiliser la complétion d'automates d'arbres, ce système de réécriture doit être linéaire à gauche.
- De plus, pour un terme $t = Res(Pre(a, Pre(b, \mathbf{0})), a)$ représentant l'expression CCS $a.b.\mathbf{0} \setminus \{a\}$, la règle de réécriture $\rho_8$ (et $\rho_1$) permet de réécrire $t$ en $t' = Sys(a, Res(Pre(b.\mathbf{0}), a))$. Le terme $t'$ représente le dérivatif $(a, \mathbf{0})$. Intuitivement on en déduit que $(a, b.\mathbf{0})$ est un dérivatif de $a.b.\mathbf{0} \setminus \{a\}$, ce qui est faux car les règles d'inférence de CCS ne permettent pas d'avoir $a.b.\mathbf{0} \setminus \{a\} \xrightarrow{a} b.\mathbf{0}$.

Pour éviter ces deux problèmes, nous allons définir les deux systèmes de réécriture $R_{sem}^{\vartheta}$ et $R_{res}^{\Theta_1,\Theta_2}$ suivants :

- Soit $R_{sem}^{\vartheta}$ le système de réécriture défini par $R_{sem}^{\vartheta} = \{\rho_1, \ldots, \rho_5\} \cup \{l\sigma \to r\sigma \mid \sigma(y) = y,\ \sigma(z) = z,\ \sigma(x) = \alpha,\ \alpha \in \vartheta,\ l \to r \in \{\rho_6, \rho_7\}\}$, où $\vartheta \subseteq \mathcal{L}$.
  L'idée ici est de substituer les variables $x$ des règles de réécriture $\rho_6$ et $\rho_7$ par les noms d'actions présents dans l'expression CCS que l'on voudrait réécrire à l'aide de $R_{sem}^{\vartheta}$.
- Soit $R_{res}^{\Theta_1,\Theta_2}$ le système de réécriture défini par $R_{res}^{\Theta_1,\Theta_2} = \{\rho_8\sigma \mid,\ \sigma(z) = z, \sigma(x) = \alpha, \sigma(y) = \beta, \alpha \in \Theta_1, \beta \in \Theta_2, \alpha \neq \beta\}$.
  On crée des règles de réécriture en substituant les variables $x$ et $y$ par des noms d'action de telle manière que $x$ soit différent de $y$. Ainsi, pour une expression CCS $E$ que l'on voudrait réécrire par $R_{res}^{\Theta_1,\Theta_2}$, on aura l'ensemble $\Theta_1 = Action(E)$ et $\Theta_2 = ResAction(E)$.
  Par exemple pour l'expression CCS $E = a.b.\mathbf{0} \setminus \{a\}$ on aura :
  $$R_{res}^{\{a,b,\mathbf{0}\},\{a\}} = \begin{array}{l}\{Res(Sys(b, p), a) \to Sys(b, Res(p, a)),\\ Res(Sys(\mathbf{0}, p), a) \to Sys(\mathbf{0}, Res(p, a))\}.\end{array}$$
  Dans la suite du document on notera $R_{\vartheta,\Theta}^{sem} = R_{sem}^{\vartheta} \cup R_{res}^{\vartheta,\Theta}$.

### 3.2.4 Transformation d'un programme CCS en un système de réécriture et un automate d'arbres

Soit un programme CCS $X = (\Lambda, \Gamma, P_0)$. On définit tout d'abord le système de réécriture $R^{\theta}_{con} = \{P \to \Phi(E) \mid (P, E) \in \theta, \theta \subseteq \mathcal{P} \times \mathcal{E}\}$. Ce dernier système de réécriture modélise les équations et la récursion.

Pour le programme CCS $X$, le système de réécriture $R_X$ est défini par $R_X = R^{sem}_{\Lambda,\Lambda'} \cup R^{\Gamma}_{con}$ avec $\Lambda' = ResAction(E) \cup ResAction(P)$ pour tout $(P, E) \in \Gamma$.

On pourra remarquer que le système de réécriture $R_X$ est non confluent (à cause des règles $\rho_2$ et $\rho_3$), et ne termine pas (à cause des règles de réécriture permettant de gérer la récursivité et lorsque pour $(P, E) \in \Gamma$ on a $P$ un sous-terme de $E$ ou de $deriv(E)$). De plus, le système de réécriture $R_X$ ne prend pas en compte les équations du chapitre 3 de [Mil89], qui permettent d'établir des égalités entre des processus. Enfin, l'automate d'arbres $A_X$ est l'automate minimal pour le langage $L_X = \{P_0\}$.

♢ **Exemple 3.9**
*Soit une expression CCS $E = (a.b.\mathbf{0} + b.\mathbf{0}) \parallel c.\mathbf{0}$. D'après la sémantique de CCS on a la transition $E \xrightarrow{a} b.\mathbf{0} \parallel c.\mathbf{0}$, prouvée par les règles d'inférence* $\mathbf{Com_1}$, $\mathbf{Sum_1}$ *et* $\mathbf{Act}$. *Avec le système de réécriture $R^{Action(E)}_{sem}$, le terme $\Phi(E)$ est réécrit en $Sys(a, (Com(Pre(b, \mathbf{0}), Pre(c, \mathbf{0}))))$ de la manière suivante :*

$$\begin{aligned} \Phi(E) &\to_{\rho_1} Com(Sum(Sys(a, Pre(b, \mathbf{0})), Pre(b, \mathbf{0})), Pre(c, \mathbf{0})) \\ &\to_{\rho_2} Com(Sys(a, Pre(b, \mathbf{0})), Pre(c, \mathbf{0})) \\ &\to_{\rho_4} Sys(a, Com(Pre(b, \mathbf{0}), Pre(c, \mathbf{0}))) \end{aligned}$$

*De manière intuitive, on peut voir qu'il est possible de faire un parallèle entre la preuve avec les règles d'inférence et la réécriture. Effectivement, on peut voir que l'application de $\rho_1$ correspond à la règle d'inférence* $\mathbf{Act}$, *la règle $\rho_2$ à* $\mathbf{Sum_1}$, *et la règle $\rho_4$ à* $\mathbf{Com_1}$.

### 3.2.5 Accessibilité de dérivatifs

Dans ce paragraphe nous allons montrer que $R^*_X(L(A_X))$ représente l'ensemble des dérivatifs du processus $P_0$, pour un programme CCS $X = (\Lambda, \Gamma, P_0)$. Nous allons tout d'abord montrer que, si $E$ a pour dérivatif immédiat $(\alpha, E')$, alors on a bien $Sys(\alpha, \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E))$ (où $\vartheta = Action(E)$ et $\Theta = ResAction(E)$). Pour cela on aura tout d'abord besoin de démontrer que si $\alpha.E_s$ est un sous-terme de $E$, alors $Pre(\alpha, \Phi(E_s))$ est bien un sous-terme de $\Phi(E)$.

♦ **Proposition 3.10**
*Soient $E$ et $E'$ deux expressions CCS soit $\alpha \in Act$. Soient deux ensembles $\vartheta = Action(E)$ et $\Theta = ResAction(E)$. Si $E \xrightarrow{\alpha} E'$ sans utiliser la règle d'inférence* $\mathbf{Con}$, *et $\alpha \in \mathcal{A} \cup \{\tau\}$, alors*

$$Sys(\alpha, \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E)).$$

*Si $E \xrightarrow{\alpha} E'$ et $\alpha \in \bar{\mathcal{A}}$, alors*

$$Sys(bar(\alpha'), \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E)), \quad \text{avec } \bar{\alpha}' = \alpha.$$

*Démonstration.* La preuve est par induction sur la structure de $E$.

**Cas 0 :** $E = \mathbf{0}$ ou $E = P$ avec $P \in \mathcal{P}$.
L'implication est vraie la prémisse étant fausse (par hypothèse on n'utilise pas la règle **Con**).

**Cas 1 :** $E = \beta.E_1$.
Comme $E \xrightarrow{\alpha} E'$, on a $\beta = \alpha$ et $E_1 = E'$. Si $\alpha \in \mathcal{A} \cup \{\tau\}$, alors, $\Phi(E) = Pre(\alpha, \Phi(E'))$. En utilisant $\rho_1$, on a alors $\Phi(E) \to_{\rho_1} Sys(\alpha, \Phi(E'))$, et donc $Sys(\alpha, \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E))$.

Si $\alpha \in \bar{\mathcal{A}}$, alors, de même, $\Phi(E) \rightarrow_{\rho_1} Sys(bar(\alpha'), \Phi(E'))$, et donc $Sys(bar(\alpha'), \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E))$.

**Cas 2 :** $E = E_1 + E_2$.

Dans ce cas, comme $E \xrightarrow{\alpha} E'$, soit $(\alpha, E')$ est un dérivatif de $E_1$, soit $(\alpha, E')$ est un dérivatif de $E_2$. Traitons le cas où $(\alpha, E')$ est un dérivatif de $E_1$ (l'autre cas est similaire, en utilisant $\rho_3$ à la place de $\rho_2$) : par hypothèse d'induction, $Sys(\alpha, \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E_1))$. Or $\Phi(E) = Sum(\Phi(E_1), \Phi(E_2))$, donc $Sum(Sys(\alpha, \Phi(E')), \Phi(E_2) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E))$. D'où, en utilisant $\rho_2$, $Sys(\alpha, \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E))$.

**Cas 3 :** $E = E_1 \parallel E_2$.

Dans ce cas, comme $E \xrightarrow{\alpha} E'$, soit (i) $\alpha \neq \tau$ et soit (i.a) $(\alpha, E')$ est un dérivatif de $E_1$, soit (i.b) $(\alpha, E')$ est un dérivatif de $E_2$, soit (ii) $\alpha = \tau$ et soit (ii.a) il existe $E'_1$, $E'_2$ et $\beta \in \mathcal{A}$ tels que $(\beta, E'_1)$ est un dérivatif de $E_1$ et $(\bar{\beta}, E'_2)$ est un dérivatif de $E_2$ et $E' = E'_1 \parallel E'_2$, soit (ii.b) il existe $E'_1$, $E'_2$ et $\beta \in \mathcal{A}$ tels que $(\bar{\beta}, E'_1)$ est un dérivatif de $E_1$ et $(\beta, E'_2)$ est un dérivatif de $E_2$ et $E' = E'_1 \parallel E'_2$. Les deux premières possibilités dans (i) se traitent de façon similaire au **Cas 2**, mais en utilisant les règles $\rho_4$ et $\rho_5$. Les deux dernières possibilités dans (ii) sont duales, on ne traitera que la situation : il existe $E'_1$, $E'_2$ et $\beta \in \mathcal{A}$ tels que $(\beta, E'_1)$ est un dérivatif de $E_1$ et $(\bar{\beta}, E'_2)$ est un dérivatif de $E_2$ et $E' = E'_1 \parallel E'_2$. On a $\Phi(E) = Com(\Phi(E_1), \Phi(E_2))$. Or, par hypothèse d'induction, $Sys(\beta, \Phi(E'_1)) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E_1))$. De même $Sys(bar(\beta), \Phi(E'_2)) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E_2))$. Donc

$$Com(Sys(\beta, \Phi(E'_1)), Sys(bar(\beta), \Phi(E'_2))) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E)).$$

D'où, en utilisant la règle obtenue à partir de $\rho_6$ en substitutant $x$ par $\beta$,

$$Sys(\tau, Com(\Phi(E'_1), \Phi(E'_2))) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E)).$$

Comme $\Phi(E') = Com(\Phi(E'_1), \Phi(E'_2))$ et $\alpha = \tau$, on a bien

$$Sys(\alpha, \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E)).$$

**Cas 4 :** $E = E_1 \setminus \ell$.

Dans ce cas, comme $E \xrightarrow{\alpha} E'$, $\alpha \neq \ell$ et $E'$ est de la forme $E' = E'_1 \setminus \ell$. De plus $(\alpha, E'_1)$ est un dérivatif de $E_1$. Par induction (et en supposant que $\alpha \in \mathcal{A} \cup \{\tau\}$), $Sys(\alpha, \Phi(E'_1)) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E_1))$. Or $\Phi(E) = Res(\Phi(E_1), \Phi(\ell))$. Donc

$$Res(Sys(\alpha, \Phi(E'_1)), \Phi(\ell)) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E)).$$

En utilisant la règle obtenue à partir de $\rho_8$ en substituant $x$ par $\alpha$ et $y$ par $\ell$, on obtient que $Sys(\alpha, Res(\Phi(E'_1), \Phi(\ell))) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E))$. Donc $Sys(\alpha, \Phi(E')) \in R^{sem*}_{\vartheta,\Theta}(\Phi(E))$. Le cas où $\alpha \in \bar{\mathcal{A}}$ se traite de façon similaire.

□

♦ **Proposition 3.11**
*Soit $X = (\Lambda, \Gamma, P_0)$ un programme CCS. Soient $E$ et $E'$ deux expressions CCS, soit $\alpha \in \mathcal{A} \cup \{\tau\}$ (resp. $\alpha \in \bar{\mathcal{A}}$). Si $E \xrightarrow{\alpha} E'$, alors $Sys(\alpha, \Phi(E')) \in R^*_X(\Phi(E))$ (resp. $Sys(\bar{\alpha}', \Phi(E')) \in R^*_X(\Phi(E))$, avec $\bar{\alpha}' = \alpha$).*

*Démonstration.* La preuve est par récurrence sur le nombre d'appels à la règle d'inférence **Con** pour obtenir $P \xrightarrow{\alpha} E'$. On ne traite que le cas $\alpha \in \mathcal{A} \cup \{\tau\}$, l'autre cas est dual.

S'il n'y a qu'un appel à **Con**, on fait une preuve par induction sur la structure de $E$.

**Cas 0 :** Si $E = \mathbf{0}$.
Il n'a aucun dérivatif, la proposition est donc vraie.

**Cas 1 :** Si $E \in \mathcal{P}$.
Comme $E \xrightarrow{\alpha} E'$ alors c'est la règle **Con** qui a été appliquée. Dans ce cas, il existe $E_1$ tel que $(E, E_1) \in \Gamma$ et $E_1 \xrightarrow{\alpha} E'$ sans utiliser la règle **Con**. Donc d'après la proposition 3.10, $Sys(\alpha, \Phi(E') \in R_X^*(\Phi(E_1))$. Or $P \rightarrow_{R_X} \Phi(E_1)$ car $(P, E_1) \in \Gamma$. Comme $E = P$, on a donc $Sys(\alpha, \Phi(E') \in R_X^*(\Phi(E))$.

**Cas 2 :** Si $E = E_1 + E_2$.
Comme $E \xrightarrow{\alpha} E'$ alors c'est soit la règle $\mathbf{Sum_1}$ qui a été appliquée et $E_1 \xrightarrow{\alpha} E'$, ou c'est la règle $\mathbf{Sum_2}$ qui a été appliquée et $E_2 \xrightarrow{\alpha} E'$. On peut supposer, par exemple, que la règle $\mathbf{Sum_1}$ a été appliquée et $E_1 \xrightarrow{\alpha} E'$. Par hypothèse d'induction, $Sys(\alpha, \Phi(E')) \in R_X^*(\Phi(E_1))$. Or $\Phi(E) = Sum(\Phi(E_1), \Phi(E_2))$, donc $Sum(Sys(\alpha, \Phi(E')), \Phi(E_2)) \in R_X^*(\Phi(E))$. D'où, en utilisant $\rho_2$, $Sys(\alpha, \Phi(E')) \in R_X^*(\Phi(E))$.

**Autres cas :** Similaires au **Cas 2**, en utilisant les mêmes arguments que dans la preuve de la proposition 3.10.

On va maintenant prouver par induction sur la structure de $E$, que si la proposition est vraie pour $n$ appels à **Con**, alors elle est vraie pour $n+1$ appels à **Con**, avec $n \geq 1$.

**Cas 0 :** Si $E = \mathbf{0}$.
Il n'a aucun dérivatif, la proposition est donc vraie.

**Cas 1 :** Si $E \in \mathcal{P}$.
Comme $E \xrightarrow{\alpha} E'$ alors c'est la règle **Con** qui a été appliquée. Dans ce cas, il existe $E_1$ tel que $(E, E_1) \in \Gamma$ et $E_1 \xrightarrow{\alpha} E'$ avec $n$ appels à **Con**. Par hypothèse de récurrence, $Sys(\alpha, \Phi(E') \in R_X^*(\Phi(E_1))$. Comme $(E, E_1) \in \Gamma$, $\Phi(E) \rightarrow_{R_X} \Phi(E_1)$. Donc $Sys(\alpha, \Phi(E')) \in R_X^*(\Phi(E))$.

**Cas 1 :** Si $E = E_1 + E_2$.
Comme $E \xrightarrow{\alpha} E'$ alors c'est soit la règle $\mathbf{Sum_1}$ qui a été appliquée et $E_1 \xrightarrow{\alpha} E'$, ou c'est la règle $\mathbf{Sum_2}$ qui a été appliquée et $E_2 \xrightarrow{\alpha} E'$. On peut supposer, par exemple, que $E_1 \xrightarrow{\alpha} E'$. Par hypothèse d'induction, $Sys(\alpha, \Phi(E') \in R_X^*(\Phi(E_1))$. Or $\Phi(E) = Sum(\Phi(E_1), \Phi(E_2))$, donc $Sum(Sys(\alpha, \Phi(E')), \Phi(E_2)) \in R_X^*(\Phi(E))$. D'où, en utilisant $\rho_2$, $Sys(\alpha, \Phi(E')) \in R_X^*(\Phi(E))$.

**Autres cas :** Similaires au **Cas 2**, en utilisant les mêmes arguments que dans la preuve de la proposition 3.10.

□

On peut maintenant généraliser pour des dérivatif de taille quelconque.

♦ **Proposition 3.12**
*Soit $X = (\Lambda, \Gamma, P_0)$ un programme CCS. Si $d \in deriv(P_0)$ alors $\Psi(d) \in R_X^*(P_0)$.*

*Démonstration.* La preuve se fait par une récurrence directe sur la longueur de la dérivation pour le dérivatif considéré, en utilisant pour le cas de base les propositions 3.10 et 3.11.

□

La proposition 3.12 va nous permettre de vérifier si des dérivatifs sont atteignables pour un programme CCS. Cependant, le système de réécriture correspondant à un programme CCS ne terminant pas, il faut calculer une sur-approximation des états atteignables en utilisant la technique décrite dans la section 2.4.

## 3.3 Exemple d'application : ABP

Dans cette section, il est présenté comment vérifier qu'un programme CCS modélisant le *Alternating Bit Protocol* (ABP) ne peut pas réaliser certaines séquences d'actions.

Dans la section 3.3.1 nous allons décrire ABP. Puis, dans la section 3.3.2, nous verrons comment le protocole est modélisé en un système de réécriture et un automate d'arbres à partir de sa spécification CCS. Et enfin, nous présenterons dans la section 3.3.3 les vérifications effectuées sur ABP.

### 3.3.1 Description de ABP

ABP est un protocole qui permet d'assurer la transmission de messages à travers un canal corrompu, pouvant perdre ou dupliquer des informations. Plus précisément, ABP est composé de deux entités, *Sender* et *Receiver*, communicant à l'aide de deux canaux (pouvant perdre ou dupliquer des messages) appelés *Trans* et *Ack*. *Sender* envoie un message accompagné du bit $b$ par le canal *Trans*, et le renvoie une ou plusieurs fois jusqu'à ce que *Receiver* envoie un accusé de réception accompagné du bit $b$ par la canal *Ack*. Après la réception de cet accusé de réception par *Sender*, ce dernier envoie (une ou plusieurs fois) un autre message accompagné du bit $b-1$ (aussi écrit $\hat{b}$) jusqu'à ce qu'il reçoive un accusé de réception avec le bit $\hat{b}$. Et ainsi de suite.

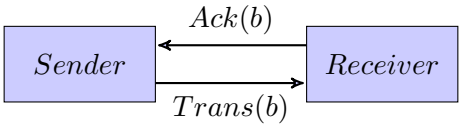



FIGURE 3.5 – Le protocole du bit alterné (ABP)

### 3.3.2 Modélisation de ABP

Ici, la spécification CCS modélisant ABP est issue de [Mil89]. Elle est représentée par le programme CCS $ABP = (\Lambda, \Gamma, AB)$ où :

- l'ensemble $\Lambda = \{accept, \overline{deliver}, ack(b), \overline{ack(b)}, reply(b), \overline{reply(b)}, send(b), \overline{send(b)}, trans(b), \overline{trans(b)} \mid b \in \{0,1\}\}$ ;
- l'ensemble $\Gamma$ est composé des règles des figures 3.6 et 3.7, où pour chaque transition $A \xrightarrow{\alpha} B$ on a $(A, \alpha.B) \in \Gamma$, avec $A$, $B \in \mathcal{E}$ et $\alpha \in \Lambda$.
  Le système de réécriture $R_{ABP}$ et le langage d'arbres $L_{ABP}$ sont définis d'après les définitions de la section 3.2. Cependant, il faut aussi compléter $R_{ABP}$ afin de gérer les séquences de bits nécessaires à la modélisation des transitions de la figure 3.7 où $b$ est un bit et $s$, $t$ sont des suites de bits sur $\{0,1\}^*$. Ainsi les symboles $Ack$ et $Trans$ seront d'arité 1 (et non 0 comme prévu par les définitions de la section 3.2 et auront en paramètre une suite de bits. Les symboles $ack$, $reply$, $send$ et $trans$ seront aussi d'arité 1 et auront en paramètre un bit. Soit le symbole $b^0$ représentant un bit $b \in \{0,1\}$ et le symbole $inv^1$ tel que $inv(b)$ représente $\hat{b}$ et on a $\hat{b} = b - 1$. On ajoute à $R_{ABP}$ la règle de réécriture $inv(inv(x)) \to x$ (où $x$ est une variable) afin de modéliser le fait que $\hat{\hat{b}} = b$.

### 3.3.3 Vérification de ABP

Dans cette section nous allons nous intéresser à la vérification de la propriété suivante : *Est-ce que ABP peut envoyer un message accompagné du bit b après avoir reçu un accusé de réception accompagné du bit b ?*

$$
\begin{array}{lcl}
Send(b) & \stackrel{def}{=} & \overline{send(b)}.Sending(b) \\
Sending(b) & \stackrel{def}{=} & \tau.Send(b) + (ack(b).Accept(\hat{b}) + ack(\hat{b}).Sending(b)) \\
Accept(b) & \stackrel{def}{=} & accept.Send(b) \\
Reply(b) & \stackrel{def}{=} & \overline{reply(b)}.Replying(b) \\
Replying(b) & \stackrel{def}{=} & \tau.Reply(b) + (trans(\hat{b}).Deliver(\hat{b}) + trans(b).Replying(b)) \\
Deliver(b) & \stackrel{def}{=} & \overline{deliver}.Reply(b) \\
AB & \stackrel{def}{=} & Accept(\hat{b}) \parallel (Trans(\varepsilon) \parallel (Ack(\varepsilon) \parallel Reply(b)))
\end{array}
$$

FIGURE 3.6 – Équations pour la définition de $ABP$

$$
\begin{array}{llllll}
Ack(bs) & \stackrel{\overline{ack(b)}}{\longrightarrow} & Ack(s) & Trans(sb) & \stackrel{\overline{trans(b)}}{\longrightarrow} & Trans(s) \\
Ack(s) & \stackrel{reply(b)}{\longrightarrow} & Ack(sb) & Trans(s) & \stackrel{send(b)}{\longrightarrow} & Trans(bs) \\
Ack(sbt) & \stackrel{\tau}{\rightarrow} & Ack(st) & Trans(tbs) & \stackrel{\tau}{\rightarrow} & Trans(ts) \\
Ack(sbt) & \stackrel{\tau}{\rightarrow} & Ack(sbbt) & Trans(tbs) & \stackrel{\tau}{\rightarrow} & Trans(tbbs)
\end{array}
$$

où $s, t \in \{0,1\}^*$ et $b \in \{0,1\}$.

FIGURE 3.7 – Transitions pour la définition de $ABP$

Pour répondre à cette question nous allons procéder de la manière suivante : dans un premier temps cette propriété doit être modélisée sous la forme d'un automate d'arbres $A_p$ reconnaissant un dérivatif du type $(\alpha^*(ack(b)\overline{send(b)}\alpha^*, E)$, où $\alpha \in \Lambda$ et $E$ est une expression CCS quelconque. Ensuite, il faut définir une fonction d'approximation permettant de calculer une sur-approximation $K$ ($\supseteq R^*_{ABP}(L_{ABP})$) de l'ensemble des dérivatifs atteignables par ABP. Enfin, on pourra utiliser l'outils TomedTimbuk [BBGM08] pour vérifier si l'intersection $K \cap L(A_p)$ est vide ou non.

A propos de la fonction d'approximation, celle-ci est définie d'après l'idée suivante : les actions prenant part à la définition de la propriété sont abstraites en un état chacune, et les autres actions sont abstraites en un seul et même état. On a la fonction d'approximation $\gamma$ telle que pour toute règle de réécriture $l \to r$ de $R_{ABP}$, pour toute substitution $\sigma$, tout état $q$ et toute position $p \in \mathcal{P}os(r)$ :

- si $r(p) \in \{b, inv\}$ alors $\gamma(l \to r, \sigma, q)(p) = q_b$,
- si $r(p) = send$ alors $\gamma(l \to r, \sigma, q)(p) = q_{send}$,
- si $r|p = bar(x)$ et $\sigma(x) = q_{send}$ alors $\gamma(l \to r, \sigma, q)(p) = q_{\overline{send}}$ (au nom de la variable prêt),
- si $r(p) = ack$ alors $\gamma(l \to r, \sigma, q)(p) = q_{ack}$,
- si $r(p) \in \{accept, reply, trans, deliver, nil, Sending, Send, Accept, Reply, Replying, Deliver\}$ alors $\gamma(l \to r, \sigma, q)(p) = q_{rem}$,
- si $r|p = bar(x)$ et $\sigma(x) = q_{rem}$ alors $\gamma(l \to r, \sigma, q)(p) = q_{\overline{rem}}$ (au nom de la variable prêt),

Enfin, à partir de l'automate d'arbres reconnaissant $L_{ABP}$, du système de réécriture $R_{ABP}$, de l'automate d'arbres $A_p$ et de la fonction d'approximation, TomedTimbuk calcul un point fixe $A_k$. L'intersection entre $L(A_k)$ et $L(A_p)$ est vide, et d'après la proposition 3.13, on peut conclure que le programme CCS $ABP$ ne peut pas réaliser une action $\overline{send(b)}$ après une action $ack(b)$.

**♦ Proposition 3.13**
*Soit $X = (\Lambda, \Gamma, P_0)$ un programme CCS, soit $L_p$ un langage représentant le dérivatif $(\alpha_0 \dots \alpha_n, E)$ avec $\alpha_0, \dots, \alpha_n \in \Lambda$ et $E \in \mathcal{E}$ tel que $L_p = \{\Psi((\alpha_0 \dots \alpha_n, E))\}$. On a : $R^*_X(L_X) \cap L_p = \emptyset$ si est*

*seulement si* $(\alpha_0 \dots \alpha_n, E)$ *n'est pas un dérivatif de* $P_0$.

Afin de prouver la proposition 3.13, on a besoin du lemme suivant.

**▲ Lemme 3.14**
*Soient* $E$ *et* $E'$ *deux expressions CCS, soient deux ensembles* $\vartheta = Action(E)$ *et* $\Theta = ResAction(E)$. *Soit* $\alpha$ *un nom d'action, on a :*

$$si\ \Phi(E) \rightarrow_{R^{sem*}_{\vartheta,\Theta}} \Psi((\alpha, E'))\ alors\ E \xrightarrow{\alpha} E'.$$

*Démonstration.* On doit montrer que $E'$ peut être construit d'après les règles d'inférence de la figure 3.2 à partir de $E$.

Comme $\mathcal{P}os_{\{Sys\}}(\Phi(E)) = \emptyset$, on a $\Phi(E) \rightarrow_{\rho_1} t_1 \rightarrow^*_{R^{sem*}_{\vartheta,\Theta}} \Psi((\alpha, E'))$ tel qu'il existe $p \in \mathcal{P}os(t_1)$ où $\Phi(E)|_p = Pre(\alpha, \Phi(E_1))$ et $t_1|_p = Sys(\alpha, \Phi(E_1))$. Si $p = \epsilon$ alors on a $\Phi(E) \equiv \Phi(\alpha.E')$, et on peut en déduire que $E \xrightarrow{\alpha} E'$ d'après la règle d'inférence **Act**. Sinon, on a $\alpha.E_1 \xrightarrow{\alpha} E_1$.

Ensuite, on argumente en fonction du terme à la position $p'$, tel que $p = p'.1$ ou $p = p'.2$ :

**Cas 1 :** $t_1|_p = Sum(Sys(\alpha, \Phi(E_1)), t_2)$ (resp. $t_1|_p = Sum(t_2, Sys(\alpha, \Phi(E_1))))$
D'après la règle de réécriture $\rho_2$ (resp. $\rho_3$), il s'ensuit que $t_1|_p \rightarrow_{\rho_2} Sys(\alpha, \phi(E_1))$ (resp. $t_1|_p \rightarrow_{\rho_3} Sys(\alpha, \Phi(E_1))$). Comme $\alpha.E_1 \xrightarrow{\alpha} E_1$, donc $\alpha.E_1 + E_2 \xrightarrow{\alpha} E_1$ (où $t_2 = \Phi(E_2)$), d'après les règles d'inférence $\mathbf{Sum_1}$ et $\mathbf{Sum_2}$.

**Cas 2 :** $t_1|_p = Com(Sys(\alpha, \Phi(E_1)), t_2)$ (resp. $t_1|_p = Com(t_2, Sys(\alpha, \Phi(E_1))))$
Similaire au Cas 1.

□

Nous allons maintenant prouver la proposition 3.13.

*Démonstration.* On doit prouver que $(R^*_X(L_X) \cap L_p = \emptyset) \Leftrightarrow ((\alpha_0 \dots \alpha_n, E) \notin deriv(P_0))$. La preuve est divisée en deux parties, on va prouver que :

**(1)** $(R^*_X(L_X) \cap L_p = \emptyset) \Rightarrow ((\alpha_0 \dots \alpha_n, E) \notin deriv(P_0))$, puis

**(2)** $((\alpha_0 \dots \alpha_n, E) \notin deriv(P_0)) \Rightarrow (R^*_X(L_X) \cap L_p = \emptyset)$.

**Preuve de (1) :** D'après la proposition 3.12 on a $d \in deriv(P_0) \Rightarrow \Psi(d) \in R^*_X(L_X)$ et donc

$$d \in deriv(P_0) \Rightarrow R^*_X(L_X) \cap \{\Psi(d)\} \neq \emptyset$$

or si on a $\neg(R^*_X(L_X) \cap \{\Psi(d)\} \neq \emptyset)$, soit $R^*_X(L_X) \cap \{\Psi(d)\} = \emptyset$, on a par contraposée

$$R^*_X(L_X) \cap \{\Psi(d)\} = \emptyset \Rightarrow d \notin deriv(P_0).$$

Ainsi le point **(1)** est prouvé.

**Preuve de (2) :** On suppose que $((\alpha_0 \dots \alpha_n, E) \notin deriv(P_0)) \Rightarrow (R^*_X(L_X) \cap L_p = \emptyset)$ est faux, on a l'hypothèse suivante : $((\alpha_0 \dots \alpha_n, E) \notin deriv(P_0)) \wedge (R^*_X(L_X) \cap L_p \neq \emptyset)$. Si $R^*_X(L_X) \cap L_p \neq \emptyset$ alors $\Psi(P_0) \rightarrow^*_{R_X} \Psi((\alpha_0 \dots \alpha_n, E))$. On va prouver que $(\alpha_0 \dots \alpha_n, E) \in deriv(P_0)$ qui est en contradiction avec l'hypothèse.

En utilisant le lemme 3.14, on a $(\alpha_0 \dots \alpha_n, E) \in deriv(P_0)$ si $\Psi(P_0) \rightarrow^*_{R_X} \Psi((\alpha_0 \dots \alpha_n, E))$. Ce qui contredit l'hypothèse et prouve **(2)**.

Enfin, d'après les preuves de **(1)** et de **(2)**, on peut conclure que $(R^*_X(L_X) \cap L_p = \emptyset) \Leftrightarrow ((\alpha_0 \dots \alpha_n, E) \notin deriv(P_0))$. □

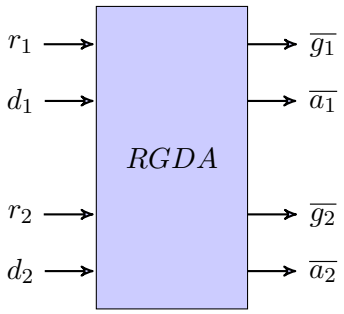



FIGURE 3.8 – Schéma du composant *RGDA*

$$
\begin{array}{lcl}
U_1 & \stackrel{def}{=} & r_1.gS.\overline{g_1}.d_1.\overline{pS}.\overline{a_1}.U_1 \\
U_2 & \stackrel{def}{=} & r_2.gS.\overline{g_2}.d_2.\overline{pS}.\overline{a_2}.U_2 \\
S & \stackrel{def}{=} & (\overline{gS}.pS.S) \setminus \{gS, pS\}
\end{array}
$$

FIGURE 3.9 – Équations pour la définition du composant *RGDA*

## 3.4 Autre exemple d'application : le composant *RGDA*

Dans cette partie nous allons nous intéresser à la vérification de propriétés pour un composant électronique modélisé à l'aide de CCS [SABL93].

Le composant *RGDA* (*Request Grant Done Acknoledgment*, figure 3.8) permet de gérer l'accès de deux utilisateurs à une section critique. Il assure qu'une seule personne accède à cette section critique. Le programme CCS correspondant à ce composant est composé des équations de la figure 3.9, où $S$ est le processus initial, $U_1$ et $U_2$ sont les utilisateurs.

La propriété que l'on veut vérifier est la suivante : *Est-ce que* RGDA *peut réaliser les actions* $\overline{g_1}$ *et* $\overline{g_2}$ *à la suite ?* De la même manière que pour le composant *Lockable*, on définit un système de réécriture $R_{RGDA}$, un automate d'arbres $A_{RGDA}$, une fonction d'approximation, et un automate d'arbres $A_p$ reconnaissant les dérivatifs du type $(\alpha^*(\overline{g_1 g_2})\alpha^*, E)$ (où $\alpha$ est un nom d'action et $E$ une expression CCS). A l'aide de TomedTimbuk, on peut déterminer que $R^*_{RGDA}(L(A_{RGDA})) \cap L(A_p)$ est vide, et on peut donc répondre *Non* à la question.

## 3.5 Conclusion

Dans ce chapitre, nous avons présenté une méthode de traduction d'un programme CCS en système de réécriture et en automate d'arbres. En utilisant la complétion d'automate d'arbres, on peut calculer une sur-approximation des dérivatifs atteignables, modulo bisimulation. Ensuite on peut vérifier si des dérivatifs non désirables ne sont pas atteignables.

Sachant qu'il existe d'autres algèbres de processus comme CSP, BPP, BPA, PA, SDL, LOTOS, ..., partageant des éléments de la syntaxe et de la sémantique de CCS, on peut imaginer adapter la réécriture de sur-approximation à ces algèbres de processus.

Pour construire cette sur-approximation, il faut construire la fonction d'approximation la plus pertinente possible. C'est-à-dire que cette fonction doit permettre permettre la terminaison du calcul de la sur-approximation sans pour autant introduire de termes inatteignables qui empêcheraient de conclure. Nous avons pu voir dans les expérimentations des sections 3.3 et 3.4 que nous pouvons générer une fonction d'approximation automatiquement en fonction de la propriété à vérifier qui mène à une analyse conclusive. Cependant, ce n'est pas toujours le

cas. Nous pouvons noter que la génération automatique de fonction d'approximation à déjà été utilisée pour la vérification de protocoles [Boi06]. Néanmoins, la définition de la fonction d'approximation nécessite une bonne connaissance du système étudié et reste difficile la plupart du temps.

# 4
# Machines de Turing

**Sommaire**



Dans ce chapitre, toujours dans l'optique d'évaluer la technique de réécriture d'approximations, nous allons aborder différents problèmes indécidables : le problème de vacuité et le problème d'appartenance d'un mot au langage d'une machine de Turing.

Tout d'abord nous donnons les définitions formelles en rapport avec les machines de Turing et de ces deux problèmes (section 4.1). Nous proposons une modélisation par systèmes de réécriture et automates d'arbres utile pour la vérification de ces deux problèmes par analyse d'accessibilité (sections 4.2, 4.3 et 4.4). Nous nous intéressons ensuite à l'approche par approximation pour semi-décider ces deux problèmes (sections 4.5 et 4.6). Pour une classe de problèmes particuliers, nous développons une procédure qui transforme automatiquement une instance du problème en un problème d'accessibilité (section 4.7). Cette procédure est expérimentée avec l'outil de manipulation d'automates d'arbres, Timbuk [GVTT01].

## 4.1 Deux problèmes sur les machines de Turing

Parmi les différents problèmes d'accessibilité indécidables, en relation avec des problématiques de vérification, nous nous intéressons au problème de vacuité du langage d'une machine de Turing et au problème d'appartenance d'un mot au langage d'une machine de Turing.

Nous allons tout d'abord définir formellement une machine de Turing ainsi que les éléments théoriques utiles à la compréhension de ce chapitre.

△ **Définition 4.1** (Machine de Turing)
*Une machine de Turing* $M$ *est un septuplet* $(Q, \Sigma, \Gamma, \#, q_0, \delta, F)$ *où :*

- $Q$ *un ensemble fini d'états ;*
- $\Sigma$ *un alphabet fini ;*
- $\Gamma$ *l'alphabet de ruban* ($\Sigma \subsetneq \Gamma$) *;*
- $\# \in \Gamma \setminus \Sigma$ *un symbole spécial (dit blanc) ;*
- $q_0 \in Q$ *l'état initial ;*
- $F \subseteq Q$ *l'ensemble des états d'acceptation ;*
- $\delta \subseteq Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ *une relation de transition.*

Une *configuration* d'une machine de Turing $M$ est définie par $c = (w_1, q, w_2)$ où $w_1, w_2 \in \Gamma^*$ sont les mots du ruban respectivement à gauche et à droite de la tête de lecture, et $q \in Q$ l'état de la machine (toutes les lettres à gauche de $w_1$ sur le ruban sont des #, ainsi que toutes les lettres à droite de $w_2$). L'ensemble des configurations d'acceptation, noté $C_{accept}$, est l'ensemble des configurations qui sont dans $\Gamma^* \times Q_f \times \Gamma^*$.
On définit maintenant la relation $\vdash_M$ sur les configurations de machines de Turing. Soit $(w_1, q, w_2)$ une configuration de la machine $M$. On pose $w_1 = w_1'\beta$ avec $\beta$ dernière lettre de $w_1$ si $w_1$ est non vide. Sinon on pose $\beta = \#$ et $w_1' = \epsilon$. De même on pose $w_2 = \alpha w_2'$ avec $\alpha$ première lettre de $w_2$ si $w_2$ est non vide. Sinon on pose $\alpha = \#$ et $w_2' = \epsilon$.

- Si $\delta(q, \alpha) = (q', \alpha', R)$, alors $(w_1, q, \alpha w_2') \vdash_M (w_1\alpha', q', w_2')$.
- Si $\delta(q, \alpha) = (q', \alpha', L)$, alors $(w_1'\beta, q, \alpha w_2') \vdash_M (w_1', q', \beta\alpha' w_2')$.

Pour deux configurations $c$ et $c'$ de $M$, on dit que $c'$ est atteignable depuis $c$, noté $c \vdash_M^* c'$, si $c = c'$ ou s'il existe une suite de configurations $c_1, ..., c_n$ telle que $c = c_1 \vdash_M c_2 \ldots \vdash_M c_n = c'$.

Un calcul de $M$ sur un mot $w$ est une suite (finie ou infinie) de configurations $c_0 \vdash_M c_1 \vdash_M c_2 \ldots \vdash_M c_n \vdash_M \ldots$ avec $c_0 = (\epsilon, q_0, w)$ la configuration initiale et tel que pour tout $i \geq 0$, $c_i \vdash_M c_{i+1}$. Le langage $L(M)$ accepté par une machine $M$ est $L(M) = \{w \in \Sigma^* \mid \exists c_{accept} \in C_{accept}$ tel que $(\epsilon, q_0, w) \vdash_M^* c_{accept}\}$.

Nous allons maintenant définir les problèmes d'appartenance d'un mot au langage reconnu par une machine de Turing et de vacuité d'un langage reconnu par une machine de Turing.

△ **Définition 4.2** (Problème d'appartenance (PAML))
**Données :** *Une machine de Turing* $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$ *et un mot* $w \in \Sigma^*$.
**Question :** Est-ce que le mot $w$ appartient à $L(M)$ ?

△ **Définition 4.3** (Problème de vacuité (PV))
**Donnée :** *Une machine de Turing* $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$.
**Question :** Est-ce que $L(M)$ est l'ensemble vide ?

Les problèmes ci-dessus sont indécidables. Nous allons maintenant aborder ces deux problèmes à l'aide de la réécriture. Dans notre étude en relation avec la problématique de vérification, nous nous intéressons aux machines de Turing déterministes.

## 4.2 Encodage en système de réécriture

Dans cette section, nous présentons une manière de modéliser les configurations et le fonctionnement d'une machine de Turing avec des termes et un système de réécriture. L'objectif est d'obtenir un système de réécriture qui nous permettra par la suite de traduire les deux problèmes (PAML et PV) en problèmes d'atteignabilité en réécriture.

On rappelle que le problème d'atteignabilité en réécriture peut s'exprimer comme suit. Soit un système de réécriture $R$, et deux termes $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ : a-t-on $s \to_R^* t$ ?

Soit $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$ une machine de Turing avec :

- $Q = \{q_0, ..., q_n\}$ ;
- $\Sigma \subseteq \Gamma$ ;
- $\Gamma = \{\alpha_0, ..., \alpha_m\} \cup \{\#\}$.

Nous proposons de modéliser ses configurations par des termes. Soient les ensembles de symboles $\mathcal{F}_0 = \{Nil\} \cup \Gamma$, $\mathcal{F}_2 = \mathcal{Q} \cup \{cons\}$ et $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_2$. La configuration $(\alpha_1 ... \alpha_i, q, \alpha_{i+1} ... \alpha_n)$ est représentée par le terme :

$$q(cons(\alpha_i, ...cons(\alpha_1, Nil)...), cons(\alpha_{i+1}, ...cons(\alpha_n, Nil)...)).$$

Soit l'ensemble de variables $\mathcal{X} = \{y, z, t\}$, nous proposons un système de réécriture $R$ linéaire dont les règles $l \rightarrow r$, avec $l$ et $r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, représentent la relation de transition $\delta$ de $M$.

Pour chaque transition $\delta(q_i, \alpha) = (q_j, \alpha', R)$ de $M$, on définit les règles de réécriture :

**(r1) :** $q_i(z, cons(\alpha, y)) \rightarrow q_j(cons(\alpha', z), y)$,

**(r2) :** $q_i(z, Nil) \rightarrow q_j(cons(\alpha', z), Nil)$ si $\alpha = \#$.

Pour chaque transition $\delta(q_i, \alpha) = (q_j, \alpha', L)$ de $M$, on définit les règles de réécriture :

**(r3) :** $q_i(cons(z, t), cons(\alpha, y)) \rightarrow q_j(t, cons(z, cons(\alpha', y)))$,

**(r4) :** $q_i(cons(z, t), Nil) \rightarrow q_j(t, cons(z, cons(\alpha', Nil)))$, uniquement si $\alpha = \#$

**(r5) :** $q_i(Nil, cons(\alpha, y)) \rightarrow q_j(Nil, cons(\#, cons(\alpha', y)))$,

**(r6) :** $q_i(Nil, Nil) \rightarrow q_j(\epsilon, cons(\#, cons(\alpha', Nil)))$ uniquement si $\alpha = \#$.

Notons que la règle **(r2)** traite le cas particulier où le mot après la tête de lecture est vide, tandis que les règles **(r5)** et **(r6)** traitent les cas particuliers où le mot précédant la tête de lecture est vide.

♢ **Exemple 4.4**
*La configuration* $c_0 = (\epsilon, q_0, aabb)$ *s'écrit :*

$$q_0(Nil, cons(a, cons(a, cons(b, cons(b, Nil))))).$$

*La transition* $\delta(q_0, a) = (q_1, A, R)$ *se représente sous forme de la règle de réécriture :*

$$q_0(x, cons(a, y)) \rightarrow q_1(cons(A, x), y)).$$

## 4.3 Relation entre $\vdash_M$ et $\rightarrow_R$

Dans cette section, on notera $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$ une machine de Turing, $c_1$ et $c_2$ des configurations de $M$ représentées respectivement par les termes $t_1$ et $t_2$, et $R$ le système de réécriture représentant $\delta$ selon la méthode décrite dans la section 4.2.

Notre but est de savoir si depuis une configuration $c_1$, la machine de Turing $M$ peut, ou ne peut pas, atteindre une configuration $c_2$. La question peut être étendue à des ensembles de configurations $C_1, C_2$ : quelque soit $c_1 \in C_1$ existe-t-il $c_2 \in C_2$ tel que $c_1 \vdash_M^* c_2$ ?

♦ **Proposition 4.5** (relation entre $\vdash_M^*$ et $\rightarrow_R^*$)
*On a :*

$$c_1 \vdash_M^* c_2 \Leftrightarrow t_1 \rightarrow_R^* t_2$$

La preuve est une simple vérification par induction sur la forme des configurations (respectivement, des termes) et des transitions de $M$ (respectivement, des règles de $R$).

## 4.4 Encodage par automates d'arbres

Pour appliquer la méthode de vérification d'accessibilité par approximation, nous avons besoin d'automates d'arbres reconnaissant les configurations initiales ou les configurations d'acceptation. Pour une machine de Turing $M$, nous définissons donc les automates :

- $A^M_{accept}$ reconnaissant les représentations des configurations d'acceptation,
- $A^M_{Init}$ reconnaissant les représentations des configurations initiales,
- $A^M_{t_{c_0}}$ reconnaissant la représentation $t_{c_0}$ de la configuration initiale $c_0$.

De plus nous montrons que ces automates reconnaissent uniquement ces configurations.

**Représentation des configurations d'acceptation.**

Nous commençons tout d'abord par montrer comment construire un automate d'arbres reconnaissant les représentations des configurations d'acceptation d'une machine de Turing.

△ **Définition 4.6** (Automate $A^M_{accept}$ reconnaissant les représentations des configurations d'acceptation)
*Soit une machine de Turing* $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$. *On pose* $A^M_{accept} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$, *où :*

- $\mathcal{F}$ *tel que défini à la section 4.2*
- $\mathcal{Q} = \{q_L, q_c, q_f\}$,
- $\mathcal{Q}_f = \{q_f\}$,
- $\Delta = \{x \to q_c \mid x \in \Gamma\} \cup \{Nil \to q_L, cons(q_c, q_L) \to q_L\} \cup \{q_x(q_L, q_L) \to q_f \mid q_x \in F\}$.

Il est facile de montrer que $A^M_{accept}$ reconnaît uniquement les termes encodant les configurations d'acceptation.

♦ **Proposition 4.7**
*Soit une machine de Turing* $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$, *soit* $T_{accept}$ *l'ensemble de termes représentant l'ensemble des configurations d'acceptation* $C_{accept}$ *de* $M$. *Soit* $A^M_{accept} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ *l'automate reconnaissant les termes de* $T_{accept}$. *On a :*

$$t_{accept} \in T_{accept} \text{ si et seulement si } t_{accept} \in L(A^M_{accept}).$$

*Démonstration.* La preuve est décomposée en deux parties ; nous allons prouver que :

**(1)** $t_{accept} \in T_{accept} \Rightarrow t \in L(A^M_{accept})$

**(2)** $t_{accept} \in L(A^M_{accept}) \Rightarrow t_{accept} \in T_{accept}$

**Démonstration de (1) :** $t_{accept} \in T_{accept} \Rightarrow t \in L(A^M_{accept})$**.** Chaque élément de $C_{accept}$ est de la forme $(\alpha_1 \dots \alpha_i, q, \alpha_{i+1} \dots \alpha_n)$, avec chaque $\alpha_i$ élément de $\Gamma$ et $q \in F$. Donc chaque élément de $T_{accept}$ s'écrit sous la forme :

$$q(cons(\alpha_i, \dots cons(\alpha_1, Nil)\dots), cons(\alpha_{i+1}, \dots cons(\alpha_n, Nil)\dots)) \text{ (a).}$$

D'après la définition 4.6, par construction de l'automate $A^M_{accept}$ nous avons la transition $q(q_L, q_L) \to q_f$ car $q \in F$. De plus, toujours d'après la définition 4.6, tout terme de la forme $cons(\mu, \dots cons(\nu, Nil)\dots)$ où $\mu$, $\nu \in \Gamma$ peut être réduit en un état $q_L$. On en déduit que tout terme peut être réduit en un état $q_f$. On a bien $t_{accept} \in T_{accept} \Rightarrow t \in L(A^M_{accept})$.

**Démonstration de (2) :** $t_{accept} \in L(A^M_{accept}) \Rightarrow t_{accept} \in T_{accept}$**.** Nous montrerons cette proposition par l'absurde. On suppose que l'on a un terme $t_{accept} \in L(A^M_{accept})$ et $t_{accept} \notin T_{accept}$ (a).

Comme $t_{accept} \in L(A^M_{accept})$ alors il peut se réduire en un état final $q_f$ d'après un transition de la forme $q_x(q_L, q_L) \to q_f$ où $q_x$ est un état d'acceptation d'une machine de Turing. Ceci implique que le terme $t_{accept}$ est de la forme $q(t_1, t_2)$ où $q \in F$ et les termes $t_1$ et $t_2$ sont de la forme $cons(\mu, \dots cons(\nu, Nil)\dots)$ où $\mu$, $\nu \in \Gamma$. Donc les termes $t_1$ et $t_2$ représentent des mots sur $\Gamma^*$. Cependant, d'après l'hypothèse (a) et la construction d'une configuration en un terme, le terme $t_{accept}$ est de la forme $q'(t_1, t_2)$ où $q' \notin F$. Nous avons une contradiction ce qui prouve que $t_{accept} \in L(A^M_{accept}) \Rightarrow t_{accept} \in T_{accept}$.

D'après les démonstrations de **(1)** et **(2)** on en déduit que $t_{accept} \in T_{accept} \Leftrightarrow t \in L(A^M_{accept})$, la proposition 4.7 est prouvée. □

**Représentation des configurations initiales.**

Dans ce paragraphe nous montrons comment construire un automate d'arbres reconnaissant toutes les représentations des configurations initiales possibles pour une machine de Turing.

△ **Définition 4.8** (Automate $A^M_{Init}$ reconnaissant les représentations des configurations initiales)
*Soit une machine de Turing* $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$. *On pose* $A^M_{Init} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$, *où*

- $\mathcal{F}$ *tel que défini à la section 4.2,*
- $\mathcal{Q} = \{q_N, q_L, q_c, q_f\}$,
- $\mathcal{Q}_f = \{q_f\}$,
- $\Delta = \{x \to q_c \mid x \in \Sigma\} \cup \{Nil \to q_N, cons(q_c, q_N) \to q_L, cons(q_c, q_L) \to q_L, q_0(q_N, q_L) \to q_f, q_0(q_N, q_N) \to q_f\}$.

Il est facile de voir que $A^M_{Init}$ reconnaît uniquement les termes encodant les configurations initiales.

♦ **Proposition 4.9**
*Soit une machine de Turing* $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$, *soit* $T_{C0}$ *l'ensemble des termes représentant les configurations initiales de* $M$, *soit l'automate* $A^M_{Init} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ *reconnaissant les termes de l'ensemble* $T_{C0}$. *Nous avons :*

$$t_{c0} \in T_{C0} \text{ si et seulement si } t_{c0} \in L(A^M_{Init}).$$

*Démonstration.* La démonstration est semblable à celle de $A^M_{accept}$ sauf que nous remplaçons, entre autres, un état $q \in F$ par $q_0$, $\Gamma$ par $\Sigma$. Dans ce cadre, nous avons la règle $q_0(q_N, q_L) \to q_f \in \Delta$, donc le terme représentant une configuration initiale $(\epsilon, q, \alpha_{i+1} \dots \alpha_n)$ sera bien accepté par l'automate.

De plus, un raisonnement par l'absurde permets de prouver qu'une configuration non initiale n'est pas acceptée. □

**Représentation d'une configuration initiale.**

Enfin nous montrons comment construire un automate d'arbres reconnaissant la représentation d'une configuration initiale d'une machine de Turing.

△ **Définition 4.10** (Automate $A^M_{tc0}$ reconnaissant la représentation d'une configuration initiale $t_{c0}$)
*Soit une machine de Turing* $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$, *soit* $t_{c0}$ *le terme représentant la configuration initiale* $c_0 = (\epsilon, q_0, w)$ *avec* $w = a_1 \dots a_k \in \Sigma^*$ *de* $M$.
*Nous définissons l'automate* $A^M_{tc0} = (\mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta)$ *de la manière suivante :*

- $\mathcal{F}$ *tel que défini à la section 4.2,*
- $\mathcal{Q}_f = \{q_f\}$,
- $\mathcal{Q}' = \{q_N, q_f, p_1, \dots, p_k, q_1, \dots, q_k\}$, *si* $w \neq epsilon$,
- $\mathcal{Q}' = \{q_N, q_f\}$ *si* $w = \epsilon$,
- $\Delta = \{q_0(q_N, q_1) \to q_f, Nil \to q_N,\ a_i \to p_i \mid 1 \leq i \leq k\} \cup \{cons(p_k, q_N) \to q_k\} \cup \{cons(p_i, q_{i+1}) \to q_i\}$, *si* $w \neq \epsilon$,
- $\Delta = \{q_0(q_N, q_N) \to q_f, Nil \to q_N\}$ *si* $w = \epsilon$.

Il est facile de voir que $A^M_{tc0}$ reconnaît uniquement $t_{c0}$.

A l'aide des définitions précédentes et de la section 4.3 nous allons maintenant voir comment semi-décider PAML et PV par approximation d'accessibilité dans les sections 4.5 et 4.6

## 4.5 Semi-décision du PAML par approximation d'accessibilité

Afin de semi-décider PAML pour une configuration $c_0$ d'une machine de Turing $M$, nous allons utiliser la technique de complétion d'automate d'arbres (vue en section 4.2) en calculant une sur- (ou sous-) approximation des configurations atteignables par $M$. Nous avons tout d'abord le résultat suivant :

♦ **Proposition 4.11** (Décision de l'appartenance par analyse d'accessibilité)
*Soit une machine de Turing $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$ modélisée par le système de réécriture $R$, la configuration $c_0 = (\epsilon, q_0, w)$, avec $w \in \Sigma^*$, représentée par le terme $t_{c_0}$, $A^M_{accept}$ un automate reconnaissant uniquement les configurations d'acceptation, on a :*

$$w \in L(M) \Leftrightarrow R^*(\{t_{c_0}\}) \cap L(A^M_{accept}) \neq \emptyset$$

*Démonstration.* Nous allons tout d'abord démontrer que $w \in L(M) \Leftrightarrow R^*(\{t_{c_0}\}) \cap L(A^M_{accept}) \neq \emptyset$. Nous allons décomposer cette preuve en deux parties, nous allons démontrer que :

**(1)** $w \in L(M) \Rightarrow R^*(\{t_{c_0}\}) \cap L(A^M_{accept}) \neq \emptyset$,

**(2)** $R^*(\{t_{c_0}\}) \cap L(A^M_{accept}) \neq \emptyset \Rightarrow w \in L(M)$.

**Démonstration de (1) :** $w \in L(M) \Rightarrow R^*(\{t_{c_0}\}) \cap L(A^M_{accept}) \neq \emptyset$**.** Si $w \in L(M)$ alors il existe une configuration d'acceptation $c_1$ telle que $(\epsilon, q_0, w) \vdash^*_M c_1$. D'après la proposition 4.5 on peut en déduire que la relation de réécriture $t_{c_0} \rightarrow^*_R t_{c_1}$ ($t_{c1} \in R^*(\{t_{c0}\})$), où le terme $t_{c_1}$ représente la configuration $c_1$. D'après la définition 4.7 le terme $t_{c_1}$ appartient à $L(A^M_{accept})$. On en déduit que l'intersection entre $R^*(\{t_{c_0}\})$ et $L(A^M_{accept})$ est non vide car elle contient au moins le terme $t_{c_1}$, la proposition **(1)** est démontrée.

**Démonstration de (2) :** $R^*(\{t_{c_0}\}) \cap L(A^M_{accept}) \neq \emptyset \Rightarrow w \in L(M)$**.** Si $R^*(t_{c_0}) \cap L(A^M_{accept}) \neq \emptyset$, alors il existe un terme $t_{c_x}$ tel que $t_{c_x} \in R^*(\{t_{c_0}\})$ (a) et $t_{c_x} \in L(A^M_{accept})$ (b). D'après (a) on peut en déduire la relation de réécriture $t_{c_0} \rightarrow^*_R t_{c_x}$ et d'après la proposition 4.5, il existe une configuration $c_x$ (représentée par le terme $t_{c_x}$) telle que $c_0 \vdash^*_M c_x$. D'après (b), le terme $t_{c_x}$ représente une configuration d'acceptation donc $c_x$ est une configuration d'acceptation. On en conclue que $w$ appartient à $L(M)$, la proposition **(2)** est démontrée.

D'après la démonstration de **(1)** et **(2)**, on peut conclure que $w \in L(M) \Leftrightarrow R^*(\{t_{c_0}\}) \cap L(A^M_{accept}) \neq \emptyset$.

□

Le calcul de $R^*(\{t_{c_0}\})$ ne terminant pas en général, nous nous tournons vers la technique exposée dans la section 2.4 du chapitre 2 pour calculer une sur-approximation, ou sous-approximation, de l'ensemble des termes atteignables par réécriture.

Nous utilisons les éléments suivants :
- l'automate $A^M_{t_{c_0}}$ acceptant la configuration initiale $t_{c_0}$ représentant $c_0 = (\epsilon, q_0, w)$, avec $w \in \Sigma^*$
- $\alpha$ une fonction d'approximation $(A^M_{t_{c_0}}, R)$-exacte,
- une fonction d'approximation $\gamma$ telle qu'il existe un entier $N$ pour lequel l'automate $(C^R_\gamma)^N(A^M_{t_{c_0}})$ est un point fixe (vis-à-vis de la complétion). On a alors $L((C^R_\gamma)^N(A^M_{t_{c_0}})) \supseteq R^*(\{t_{c_0}\})$.

Ceci nous permet éventuellement de calculer un sur- (ou sous-) ensemble de $R^*(\{t_{c_0}\})$, et d'en déduire l'appartenance, ou non, du mot représenté par $t_{c_0}$ à $L(M)$.

★ **Théorème 4.12** (Semi-décision de la non appartenance par une sur-approximation des termes accessibles)
*S'il existe un* $N \in \mathbb{N}$ *tel que* $(C_\gamma^R)^N(A_{t_{c_0}}^M) = (C_\gamma^R)^{N+1}(A_{t_{c_0}}^M)$ *alors :*

$$L((C_\gamma^R)^N(A_{t_{c_0}}^M)) \cap L(A_{accept}^M) = \emptyset \Rightarrow w \notin L(M)$$

*Démonstration.* D'après la proposition 4.11 on a l'implication suivante :

$$R^*(\{t_{c_0}\}) \cap L(A_{accept}^M) = \emptyset \Rightarrow w \notin L(M).$$

On sait que $L((C_\gamma^R)^N(A_{t_{c_0}}^M)) \cap L(A_{accept}^M) = \emptyset$ et par définition on sait que

$$R^*(\{t_{c_0}\}) \subseteq L((C_\gamma^R)^N(A_{tc0}^M)).$$

On en déduit que $L((C_\gamma^R)^N(A_{t_{c_0}}^M)) \cap L(A_{accept}^M) = \emptyset \Rightarrow w \notin L(M)$, le théorème 4.12 est prouvé. □

★ **Théorème 4.13** (Semi-décision de l'appartenance par une sous-approximation des termes accessibles)
*S'il existe un* $N \in \mathbb{N}$ *tel que* $L((C_\alpha^R)^N(A_{t_{c_0}}^M)) \cap L(A_{accept}^M) \neq \emptyset$ *alors* $w \in L(M)$.

*Démonstration.* D'après la proposition 4.11 on a l'implication suivante :

$$R^*(\{t_{c_0}\}) \cap L(A_{accept}^M) \neq \emptyset \Rightarrow w \in L(M).$$

On sait que $L((C_\alpha^R)^N(A_{t_{c_0}}^M)) \cap L(A_{accept}^M) \neq \emptyset$ et par définition on sait que

$$R^*(\{t_{c_0}\}) \supseteq L((C_\alpha^R)^N(A_{tc0}^M)).$$

On en déduit que $L((C_\alpha^R)^N(A_{t_{c_0}}^M)) \cap L(A_{accept}^M) \neq \emptyset \Rightarrow w \in L(M)$, le théorème 4.13 est prouvé. □

## 4.6 Semi-décision du PV par approximation d'accessibilité

Pour une machine $M = (Q, \Sigma, \Gamma, \#, q_0, \delta, F)$, décider le problème de vacuité "$L(M) = \emptyset$ ?" revient à savoir décider si

$$\forall w \in \Sigma^* \; w \notin L(M).$$

Exprimée en configurations de machine de Turing la question devient, pour une instance du problème :

$$\forall w \in \Sigma^* \; \forall c_{accept} \in C_{accept} \; (\epsilon, q_0, w) \not\vdash_M^* c_{accept}.$$

Soit un système de réécriture $R$ modélisant $M$, un automate d'arbres $A_{Init}^M$ reconnaissant les configurations initiales de la machine $M$, et $A_{accept}^M$ reconnaissant les configurations d'acceptation. Si l'intersection d'une sur-approximation de $R^*(L(A_{Init}^M))$ et de $L(A_{accept}^M)$ est vide alors le langage reconnu par $M$ est vide, sinon il ne l'est pas.

Nous avons tout d'abord le résultat de décision suivant pour la résolution du problème de vacuité pour une machine de Turing.

♦ **Proposition 4.14** (Décision du vide par un calcul d'accessibilité)
*Soit la machine de Turing $M$ représentée par le système de réécriture $R$, $C_0$ l'ensemble des configurations initiales de $M$ représenté par l'ensemble de termes $T_{C_0}$, $C_{accept}$ l'ensemble des configurations d'acceptation de $M$ représenté par l'ensemble de termes $T_{C_{accept}}$, $A^M_{Init}$ un automate d'arbres reconnaissant les termes de $T_{C_0}$, $A^M_{accept}$ un automate d'arbres tel que $L(A^M_{accept}) = T_{C_{accept}}$, nous avons :*

$$L(M) \neq \emptyset \Leftrightarrow R^*(L(A^M_{Init})) \cap L(A^M_{accept}) \neq \emptyset.$$

*Démonstration.* Cette proposition est une conséquence de la proposition 4.11.

Si $L(M) \neq \emptyset$ alors il existe un mot $w \in L(M)$ ce qui implique que $R^*(L(A^M_{Init})) \cap L(A^M_{accept}) \neq \emptyset$ d'après la proposition 4.11. Si $R^*(L(A^M_{Init})) \cap L(A^M_{accept}) \neq \emptyset$ alors il existe un mot $w \in L(M)$ d'après la proposition 4.11 et donc cela implique que $L(M) \neq \emptyset$.

On en déduit que $L(M) \neq \emptyset \Leftrightarrow R^*(L(A^M_{Init})) \cap L(A^M_{accept}) \neq \emptyset$.

□

Puisque le calcul de $R^*(L(A^M_{Init}))$ ne termine pas en général, nous utilisons les éléments suivants :

- l'automate $A^M_{Init}$ reconnaissant les configurations initiales $T_{C_0}$,
- $\alpha$ une fonction d'approximation $(A^M_{Init}, R)$-exact,
- une fonction d'approximation $\gamma$ telle qu'il existe un entier $N$ pour lequel l'automate $(C^R_\gamma)^N(A^M_{Init})$ est un point fixe (vis-à-vis de la complétion). On a alors $L((C^R_\gamma)^N(A^M_{Init})) \supseteq R^*(L(A^M_{Init}))$.

Ceci nous permet éventuellement de calculer un sur- (ou sous-) ensemble de $R^*(T_{C_0})$, et de déduire si $L(M)$ est vide ou non.

★ **Théorème 4.15** (Semi-décision du vide par une sur-approximation des termes accessibles)
*S'il existe $N \in \mathbb{N}$ tel que $(C^R_\gamma)^N(A^M_{Init}) = (C^R_\gamma)^{N+1}(A^M_{Init})$ alors :*

$$L((C^R_\gamma)^N(A^M_{Init})) \cap L(A^M_{accept}) = \emptyset \Rightarrow L(M) = \emptyset$$

*Démonstration.* D'après la proposition 4.14 on a l'équivalence suivante :

$$L(M) = \emptyset \Leftrightarrow R^*(L(A^M_{Init})) \cap L(A^M_{accept}) = \emptyset.$$

Par définition on sait que $R^*(L(A^M_{Init})) \subseteq L((C^R_\gamma)^N(A^M_{Init}))$ alors on en déduit que :

$$L((C^R_\gamma)^N(A^M_{Init})) \cap L(A^M_{accept}) = \emptyset \Rightarrow L(M) = \emptyset.$$

Le théorème 4.15 est prouvé. □

Par ailleurs, nous pouvons exploiter les sous-approximations pour décider si le langage d'une machine de Turing n'est pas vide.

★ **Théorème 4.16** (Semi-décision du vide par une sous-approximation des termes accessibles)
*S'il existe $N \in \mathbb{N}$ tel que $L((C^R_\alpha)^N(A^M_{Init})) \cap L(A^M_{accept}) \neq \emptyset$ alors $L(M) \neq \emptyset$.*

*Démonstration.* D'après la proposition 4.14 on a l'équivalence suivante :

$$L(M) \neq \emptyset \Leftrightarrow R^*(L(A^M_{Init})) \cap L(A^M_{accept}) \neq \emptyset.$$

Par définition on sait que $R^*(L(A^M_{Init})) \supseteq L((C^R_\alpha)^N(A^M_{Init}))$ alors on en déduit que :

$$L((C^R_\alpha)^N(A^M_{Init})) \cap L(A^M_{accept}) \neq \emptyset \Rightarrow L(M) \neq \emptyset.$$

Le théorème 4.16 est prouvé. □

## 4.7 Exemple d'application

Dans le but de valider notre modèle pour une utilisation réelle, nous avons implanté notre modélisation de machine de Turing en réécriture.

Le calcul de complétion d'automates d'arbres est (très) fastidieux à faire manuellement. Il est impossible de faire de tels calculs pour des problèmes autres que jouets. Nous avons donc utilisé le logiciel Timbuk [GVTT01] automatisant ces calculs. Notons que la complétion est aussi implémentée dans TomedTimbuk [BBGM08].

Quelques expérimentations ont été faites sur des instanciations du problème de PAML pour une machine de Turing $M_x$ telle que que $L(M_x) = \{a^n b^n c^n | n \in \mathbb{N}\}$. Quelques résultats obtenus grâce à l'utilisation de Timbuk sont résumés dans le tableau 4.1. Ce tableau est constitué de trois colonnes :

1. mots dont on désire connaître l'appartenance ou non à $L(M_x)$ ; notons que pour les deux dernière ligne du tableau, un problème un peu plus général a été traiter, celui de savoir s'il l'un des mots du langage donnée état accepté par la machine de Turing.
2. fonctions d'approximation utilisées pour la complétion (soit une fonction $\gamma$ définie à la main, soit une fonction $(A, R)$-exacte) puis le nombre d'étapes de complétion réalisées et enfin si l'automate donné en résultat est un point fixe ($\surd$) ou non ($\times$),
3. résultat de l'intersection des langages de l'automate issu de la complétion et de l'automate reconnaissant les configurations d'acceptation de $M_x$.

| PAML | Approx. : | nb étapes | point fixe ? | Résultat |
|---|---|---|---|---|
| $abc$ | $(A, R)$-exacte : | 9 | $\surd$ | $\neq \emptyset$ |
| $a^2b^2c^2$ | $(A, R)$-exacte : | 24 | $\surd$ | $\neq \emptyset$ |
| $a^3b^3c^3$ | $(A, R)$-exacte : | 47 | $\surd$ | $\neq \emptyset$ |
| $a^4b^4c^4$ | $(A, R)$-exacte : | 78 | $\surd$ | $\neq \emptyset$ |
| $a^5b^5c^5$ | $(A, R)$-exacte : | 112 | $\surd$ | $\neq \emptyset$ |
| $a^+b^+c^+$ | $(A, R)$-exacte : | 8 | $\times$ | $\neq \emptyset$ |
| $a^+b^+$ | approx. $\gamma$ : | 4 | $\surd$ | $= \emptyset$ |

FIGURE 4.1 – Résolution de PAML pour $\{a^n b^n c^n \mid n \geq 0\}$.

Les résultats étaient prévisibles pour cette machine, et montrent bien la pertinence de la modélisation. La fonction d'approximation utilisée est ad-hoc pour ces instances du problème.

# 5 Conclusion

On a pu voir dans cette partie deux adaptations de la réécriture par complétion à des problèmes indécidables pour l'algèbre de processus CCS et pour les machines de Turing. Dans les deux cas, cette adaptation s'est faite par la traduction de la syntaxe et de la sémantique en automate d'arbres et en système de réécriture. Ensuite on a traduit en des problèmes d'atteignabilité en réécriture les problèmes :

- de vérifications de propriété d'atteignabilité pour les spécifications CCS,
- du vide et de l'appartenance d'un mot à un langage pour les machines de Turing.

Afin de pouvoir semi-décider ces problèmes, nous avons utilisé la complétion d'automate d'arbres. Cette procédure nécessite la définition d'une fonction d'approximation qui influencera sur la qualité de l'approximation calculée par la complétion, si elle termine. La définition d'une fonction d'approximation est :

- directement dépendante des règles de réécriture (de la modélisation du problème),
- faite en fonction de la propriété à vérifier (la plupart du temps).

Une bonne fonction d'approximation mène à la conclusion que l'automate complété ne reconnaît aucun terme indésirable (si possible), on peut dire que c'est une mauvaise fonction d'approximation sinon.

Ainsi, la définition par un opérateur humain de la fonction d'approximation nécessite une connaissance technique pointue du système analysé et de la méthode de complétion. Il est très difficile pour celui-ci de prévoir le résultat exact de la complétion et la qualité de l'approximation pour une fonction d'approximation. La qualité est uniquement évaluée lors du résultat du calcul de l'intersection entre le langage reconnu par l'automate complété et le langage des termes indésirables. Si l'intersection est vide alors la fonction d'approximation est bonne. Sinon la fonction est mauvaise et l'opérateur doit la modifier afin d'obtenir une intersection vide, si possible.

De plus pour un même système, si l'on veut vérifier plusieurs propriétés il faut le plus souvent définir une nouvelle fonction d'approximation pour chacune des propriétés.

Dans la suite du document nous allons voir une solution permettant la modification automatique d'une mauvaise fonction d'approximation afin d'en trouver une meilleure, voire une bonne.

Troisième partie

# Améliorations et extensions de la complétion

# 6

# Le raffinement d'approximation

**Sommaire**



## 6.1 Raffinement et model-checking régulier

Dans cette thèse, nous nous intéressons à la résolution du problème du model-checking régulier à l'aide d'une approche par sur-approximation. Soit $R$ un système de réécriture, $I$ un ensemble régulier de termes et $P$ un ensemble régulier de termes indésirables. Nous calculons, si possible, une sur-approximation $K$ de $R^*(I)$ et nous calculons son intersection avec $P$ afin de savoir si des termes de $P$ ne sont pas atteignables.

Nous avons les deux situations suivantes, comme le montre la figure 6.1 :

1. Soit nous avons $K \cap P = \emptyset$ et nous pouvons conclure que aucun terme de $P$ n'est atteignables par réécriture de $I$ par $R$ ($R^*(I) \cap P = \emptyset$).
2. Soit nous avons $K \cap P \neq \emptyset$ est nous ne pouvons pas conclure quand à l'appartenance de termes de $P$ à l'ensemble $R^*(I)$. Dans ce cas un terme de $(K \setminus R^*(I)) \cap P$ est appelée *faux positif*, alors qu'un terme de $R^*(I)) \cap P$ est appelée *vrai contre-exemple*.

Dans le cas 2. le problème est de savoir distinguer un vrai contre-exemple d'un faux positif et aussi, dans le cas où nous savons que nous sommes en présence d'un contre-exemple faux positif, comment calculer une nouvelle sur-approximation $K'$ telle que $K' \cap P = \emptyset$.

Dans ce chapitre nous exposons une méthode de raffinement d'approximation permettant, si possible, de résoudre ces deux problèmes.

Dans le contexte de la réécriture d'automate d'arbres par complétion, on a un système de réécriture $R$, une fonction d'approximation $\gamma$, et un automate d'arbres $A$ tel que $R^*(L(A_0)) \subseteq L((C_\gamma^R)^{(N)}(A))$ (en supposant que la procédure converge). Ici le raffinement d'approximation est un processus modifiant $\gamma$ automatiquement afin d'obtenir, si possible, un point fixe reconnaissant moins de terme. Ainsi on peut partir d'une fonction d'approximation initiale générée automatiquement et qui sera ensuite modifiée par raffinement, ce qui évite une définition manuelle (qui

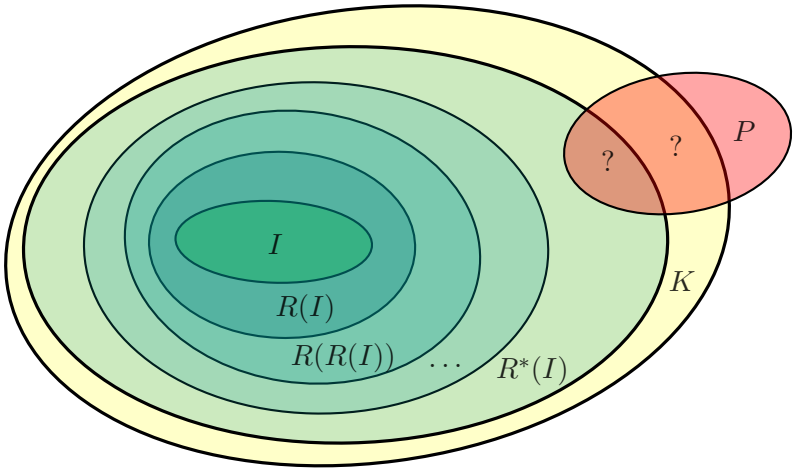



FIGURE 6.1 – Le model-checking régulier par sur-approximation

peut s'avérer laborieuse) de cette fonction d'approximation. Ce processus est décrit plus précisément dans la section 6.1.1, qui sera suivie d'un état de l'art sur le raffinement (section 6.1.2). Ensuite, on présentera les différents éléments nécessaires au raffinement d'approximation : le raffinement de la fonction d'approximation en section 6.2 et l'analyse en arrière par complétion en section 6.3. Ensuite, on verra un semi-algorithme pour le raffinement à la section 6.4 et une expérimentation à la section 6.5. Le chapitre se terminera avec un bilan et les perspectives pour le raffinement d'approximation (section 6.6).

### 6.1.1 Illustration sur les mots

Dans cette section nous allons présenter les grands principes du raffinement d'approximation afin de permettre une compréhension plus aisée des sections suivantes de ce chapitre. Tout d'abord le raffinement d'approximation est illustré à l'aide d'un automate sur les mots, et ensuite on ramènera ce problème dans le cadre de la réécriture d'automates d'arbres.

Dans cette section, on utilisera des automates de mots (tout comme dans l'exemple 2.21) pour illustrer le raffinement d'approximation.

On va s'intéresser ici à la réécriture des mots reconnus par l'automate $A_0$ (figure 6.2) par le système de réécriture

$$R = \{a(x) \to d(c(x)),\ b(x) \to f(e(x))\}.$$

Ainsi on a $R^*(L(A_0)) = \{b(a(\omega)), f(e(a(\omega))), b(d(c(\omega))), f(e(d(c(\omega))))\}$. L'ensemble des termes indésirables est ici réduit au seul terme $f(f(e(d(c(\omega)))))$, reconnu par l'automate $A_p$. On peut voir que l'instersection entre $R^*(L(A_0))$ et $L(A_p)$ est vide.

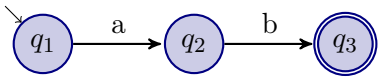



FIGURE 6.2 – Automate initial $A_0$

On réalise une première étape de complétion, il y a deux paires critiques :

$$(a(x) \to d(c(x)), \{x \to q_1\}, q_2),$$

$$(b(x) \to f(e(x)), \{x \to q_2\}, q_3).$$

On utilise la fonction d'approximation $\gamma_1$ définie de la façon suivante :

$$\gamma_1(a(x) \to d(c(x)), \{x \to q_1\}, q_2) = \{1 \to q_2\},$$

$$\gamma_1(b(x) \to f(e(x)), \{x \to q_2\}, q_3) = \{1 \to q_3\}.$$

Après cette étape de complétion on obtient l'automate $A_1 = C^R_{\gamma_1}(A_0)$ (figure 6.3) reconnaissant le langage $L(A_1) = \{f^*(\mathfrak{m}(d^*(\mathfrak{n}(\omega)))) \mid \mathfrak{m} \in \{b, e\}, \mathfrak{n} \in \{a, c\}\}$. Il n'y a aucune nouvelle paire critique, cet automate est un point fixe pour la procédure de complétion. Cependant, l'intersection entre $L(A_1)$ et $L(A_p)$ est non-vide. On ne peut donc pas conclure quand à l'appartenance ou non d'un terme de $L(A_p)$ à l'ensemble $R^*(L(A_0))$.

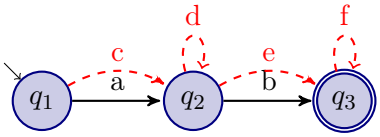



FIGURE 6.3 – Automate $A_1$

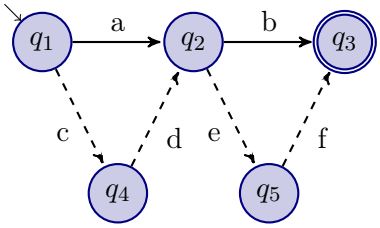



FIGURE 6.4 – Automate $A_1^e$

Comme on a $L(A_1) \cap L(A_p) \neq \emptyset$, on va calculer à nouveau la première étape de complétion, mais avec une fonction d'approximation $(A, R)-$exacte $\alpha$. On va obtenir l'automate $A_1^e$ (figure 6.4) et on vérifie que $L(A_1^e) \cap L(A_p) = \emptyset$. Cela veut dire que la fonction d'approximation $\gamma_1$ a été trop grossière et qu'il est possible de la modifier pour que l'étape de complétion n'introduise pas de terme indésirable.

On peut par exemple utiliser la fonction d'approximation $\gamma_{raff}$ définie de la façon suivante :

$$\gamma_{raff}(a(x) \to d(c(x)), \{x \to q_1\}, q_2) = \{1 \to q_2\},$$

$$\gamma_{raff}(b(x) \to f(e(x)), \{x \to q_2\}, q_4) = \{1 \to q_5\}.$$

Si on réalise une étape de complétion à partir de l'automate $A_0$ avec la fonction d'approximation $\gamma_{raff}$, on obtient un automate $A_1^{raff}$ (figure 6.13). Cet automate reconnaît le langage $L(A_1^{raff}) = \{b(d^*(a(\omega))), b(d^*(c(\omega))), f(e(d^*(a(\omega)))), f(e(d^*(c(\omega))))\}$, et on a $L(A_1^{raff}) \cap L(A_p) = \emptyset$.

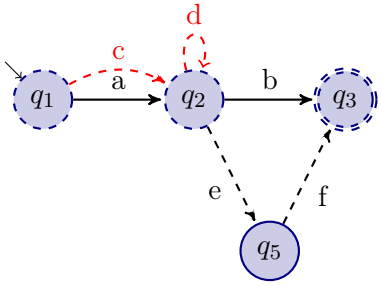



FIGURE 6.5 – Automate $A_1^{raff}$

L'objectif du raffinement d'approximation est de modifier (automatiquement) la fonction d'approximation (ici $\gamma_1$) afin d'éviter l'introduction de termes indésirables lors des étapes de complétion (pour obtenir ici $\gamma_{raff}$). On verra par la suite que cette modification n'est pas

toujours possible, ou n'est pas possible dès la première étape où des termes indésirables sont atteints (contrairement à l'exemple ci-dessus).

L'objectif de ce chapitre est d'effectuer cette démarche dans le cadre plus général de la complétion sur les automates d'arbres. Le point clé sera de savoir comment calculer $\gamma_{raff}$ à partir de $\gamma_1$.

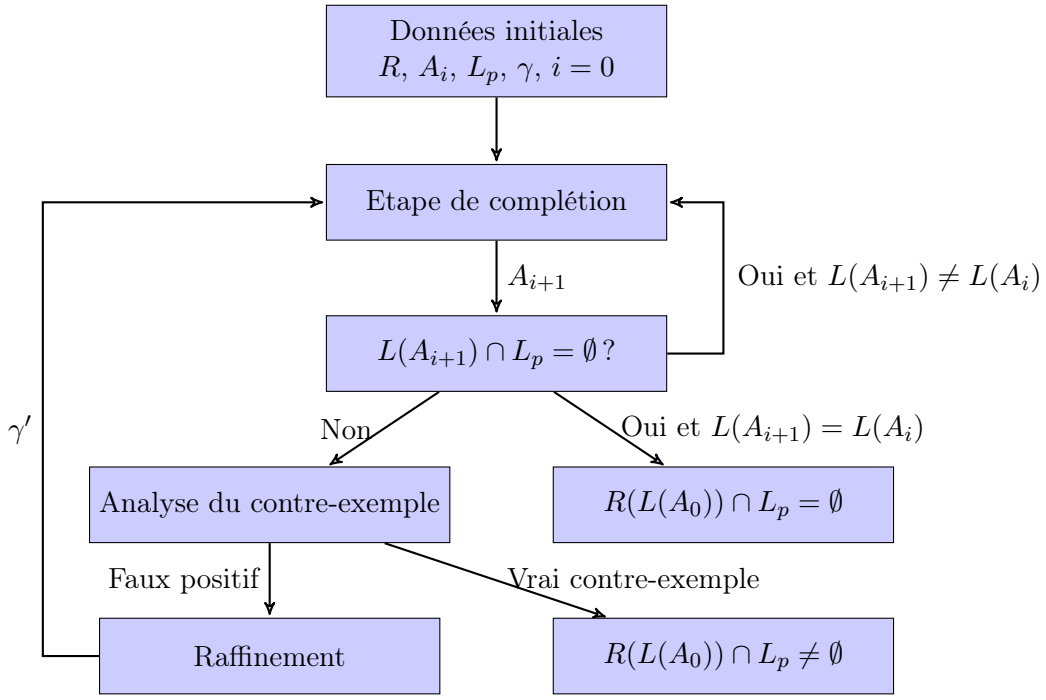



FIGURE 6.6 – Le raffinement d'approximation

Ce travail peut s'apparenter par plusieurs aspects au raffinement d'abstraction qui a déjà été étudié dans d'autres contextes comme on peut le voir dans la section 6.1.2.

### 6.1.2 État de l'art

Dans [Cla03], Clarke définit une méthode générale (*Counterexample-Guided Abstraction Refinement*, CEGAR) afin de résoudre ce problème sur des structures de Kripke en modifiant une fonction d'approximation $h$ en fonction des contre-exemples trouvés :

**(i) Initialisation.** Génération d'une fonction d'approximation initiale $h$.

**(ii) Model checking.** Vérification de propriétés sur le modèle approximé $\hat{K}$, obtenu grâce au modèle concret $K$ et à la fonction d'approximation $h$. Si la vérification réussit alors on arrête l'algorithme. Sinon on génère un contre-exemple $\hat{T}$ à partir du modèle $\hat{K}$.

**(iii) Analyse du contre-exemple.** Déterminer si $\hat{T}$ est un faux positif. Si un contre-exemple du modèle concret est trouvé à partir de $\hat{T}$ alors on arrête l'algorithme : la propriété n'est pas vérifiée.

**(iv) Raffinement.** Sinon, on raffine $h$ de tel façon que $\hat{T}$ ne soit plus un contre-exemple dans le nouveau modèle approximé et on retourne à (ii).

Comme on l'a vue précédemment, le même principe est utilisé dans ce chapitre pour le raffinement d'approximation. Le raffinement d'abstraction à partir de contre-exemples a fait

l'objet de très nombreux articles. Nous présentons ici les premiers, ainsi qu'un article dans le cadre du model-checking régulier.

Dans [CGJ+01], les auteurs s'intéressent au raffinement dans le cadre de modélisations par des structures de Kripke. Dans ce cadre, l'approximation consiste à construire une structure plus petite (en nombre d'états) en fusionnant des états de la modélisation concrète. Dans ce contexte un contre-exemple est un chemin dans la structure de Kripke. La figure 6.7 illustre la situation d'un faux positif. Les états $\hat{1}$, $\hat{2}$, $\hat{3}$, $\hat{4}$ sont des états de la structure approximante, obtenue par $h$ (on a $h(i) - 1 = i \mod 4$). Supposons que le chemin $\hat{T} = (\hat{1}\hat{2}), (\hat{2}\hat{3}), (\hat{3}\hat{4})$ viole la propriété voulue, c'est-à-dire qu'il constitue un contre-exemple. C'est un faux positif car il n'y a aucun chemin possible dans le système concret dont l'image par $h$ donne $\hat{T}$. Le raffinement d'approximation va alors constituer à utiliser une nouvelle fonction $h'$, par exemple $h'(i) = h(i)$ si $i \neq \{9, 10, 11, 12\}$, $h'(9) = h'(10) = \hat{3}$ et $h'(11) = h'(12) = \widehat{3bis}$.

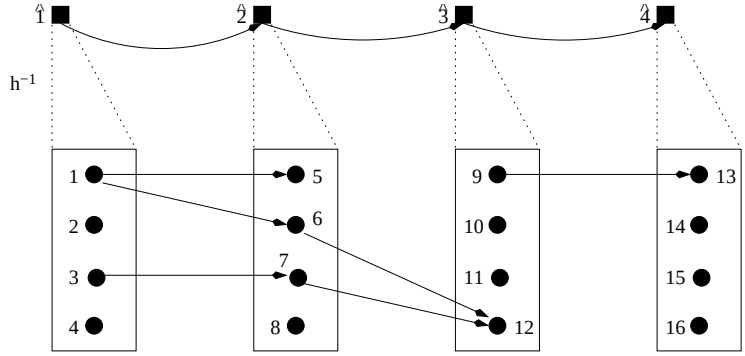



FIGURE 6.7 – Contre-exemple faux positif

Dans [LBBO01], les auteurs présentent une technique similaire, mais utilisable sur des systèmes infinis et utilisant des accélérations pour améliorer la qualité du raffinement.

Dans [BHRV06], les auteurs adaptent aux automates d'arbres une technique appelée *abstract regular model checking* (ARMC) déjà utilisée avec des automates sur les mots [BHV04b], qui devient donc *abstract regular tree model checking* (ARTMC). Cette technique entre dans le cadre du calcul d'abstraction à l'aide de prédicats appliquée au model-checking régulier. Le raffinement est obtenu à partir de contre-exemples en ajoutant de nouveaux prédicats, calculés à partir des automates utilisés.

Dans [BBGL10], les auteurs utilisent des équations à la place de fonctions d'approximation et étendent le modèle des automates d'arbres afin de distinguer des termes issus de l'approximation et des termes atteignables après avoir réécrit l'automate d'arbres initial et utilisé les équations.

On a pu voir plusieurs techniques de raffinement d'approximations intégrées à des méthodes de model checking utilisant toutes le même principe énoncé dans [Cla03]. La technique développée dans les sections suivantes utilise ce même principe mais avec des structures de données utilisées pour la complétion d'automate (automates d'arbres, systèmes de réécriture).

## 6.2 Le raffinement d'approximation dans le cadre de la complétion

Pour toutes les définitions, propositions et preuves qui suivent on a comme hypothèses : $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ un automate d'arbres et $A_P$ un automate d'arbres reconnaissant un ensemble régulier de termes indésirables, $R$ un système de réécriture, $\gamma$ une fonction d'approximation, $\alpha$ une fonction d'approximation $(A, R)-$exacte.

Le raffinement d'approximation est effectué lorsque les conditions suivantes, illustrées dans la figure 6.8, sont réunies :

**(1)** $L(C^R_\gamma(A)) \cap L(A_P) \neq \emptyset$, $L(A) \cap L(A_P) = \emptyset$

**(2)** et $L(C^R_\alpha(A)) \cap L(A_P) = \emptyset$.

Pour obtenir la condition **(1)**, on verra dans la section 6.4 que l'on applique la procédure de complétion justement tant que **(1)** n'est pas satisfaite. Dans le cas où la condition **(2)** ne serait alors pas satisfaite, on verra en section 6.3 comment procéder.

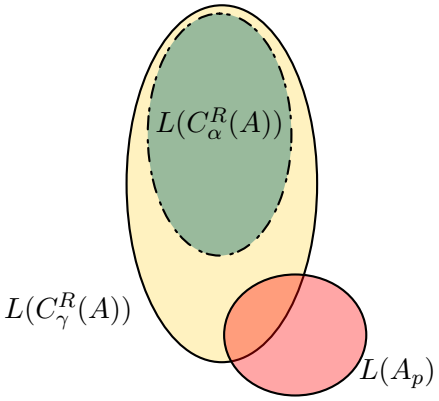



FIGURE 6.8 – $L(C^R_\alpha(A)) \cap L(A_P) = \emptyset$ et $L(C^R_\gamma(A)) \cap L(A_P) \neq \emptyset$

Ceci signifie que la fonction d'approximation $\gamma$ a permis de réaliser une sur-approximation trop grossière lors de l'étape de complétion. En effet les termes de $L(A_P)$ sont atteignables en utilisant $\gamma$, mais ne le sont pas quand on utilise une fonction d'approximation $(A, R)-$exacte. On peut en déduire que la fonction d'approximation $\gamma$ introduit des transitions problématiques, conduisant à une intersection non vide entre $L(C^R_\gamma(A))$ et $L(A_P)$, comme illustré par la figure 6.9. La définition suivante permet de rafiner une fonction d'approximation en fonction d'un ensemble de transitions problématiques.

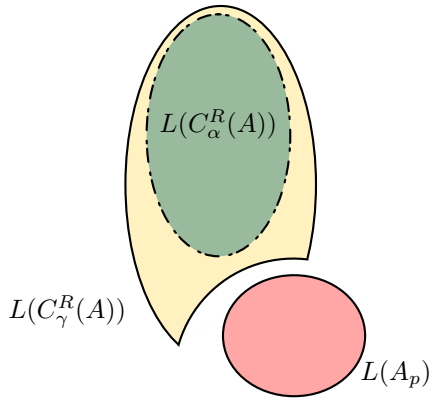



FIGURE 6.9 – $L(C^R_\alpha(A)) \cap L(A_P) = \emptyset$ et $L(C^R_\gamma(A)) \cap L(A_P) = \emptyset$

△ **Définition 6.1** (Fonction d'approximation raffinée)
*Soit une paire critique* $(l \to r, \sigma, q)$ *et une position* $p \in \mathcal{P}os_{\mathcal{F}}(r)$. *Soit* $\Delta_0$ *un ensemble de*

*transitions. On définit une fonction d'approximation raffinée* $\gamma_{\Delta_0}$ *comme suit :*

$$\gamma_{\Delta_0}(l \to r, \sigma, q)(p) \stackrel{def}{=} \begin{cases} \alpha(l \to r, \sigma, q)(p) & si \quad Norm(l \to r, \sigma, q) \cap \Delta_0 \neq \emptyset \\ \gamma(l \to r, \sigma, q)(p) & sinon \end{cases}$$

On peut maintenant définir un automate d'arbres caractérisé par une fonction d'approximation raffinée, que l'on appelle automate raffiné.

△ **Définition 6.2** (Ensemble de transitions discriminant et automate raffiné)
*Soit* $\Delta_0$ *un ensemble de transitions. On dit que* $\Delta_0$ *est discriminant si* $L(C^R_{\gamma_{\Delta_0}}(A)) \cap L(A_P) = \emptyset$.

*L'automate raffiné de* $C^R_\gamma(A)$ *est l'automate* $C^R_{\gamma_{\Delta_0}}(A)$.

Pour une fonction d'approximation $\gamma$ (respectivement $\alpha$), on note $\Delta_\gamma$ (resp. $\Delta_\alpha$) l'ensemble des transitions de $C^R_\gamma(A)$ (resp. $C^R_\alpha(A)$), mais qui ne sont pas dans l'ensemble $\Delta$ des transitions de $A$. La proposition suivante énonce le fait que, étant donné $\gamma$ et $A$, on peut raffiner $\gamma$ de façon à ce que les termes indésirables ne soient plus reconnus par l'automate d'arbres issu d'une étape de complétion sur $A$, en utilisant $R$ et une fonction d'approximation raffinée.

♦ **Proposition 6.3** (Propriété de l'automate raffiné)
*Si la condition suivante est satisfaite*

**(2)** $L(C^R_\alpha(A)) \cap L(A_P) = \emptyset$,

*alors il existe* $\Delta_0$ *tel que* $L(C^R_{\gamma_{\Delta_0}}(A)) \cap L(A_P) = \emptyset$.

*Démonstration.* On pose $C^R_\gamma(A) = (\mathcal{F}, \mathcal{Q}_\gamma, \mathcal{Q}_{f\gamma}, \Delta \cup \Delta_\gamma)$, où $\Delta_\gamma$ est l'ensemble des transitions obtenu par normalisation. On pose $\Delta_0 = \Delta_\gamma$. Dans ce cas on a $L(C^R_{\gamma_{\Delta_0}}(A)) = L(C^R_\alpha(A))$. Par hypothèse **(2)** on a $L(C^R_\alpha(A)) \cap L(A_P) = \emptyset$, donc on obtient bien $L(C^R_{\gamma_{\Delta_0}}(A)) \cap L(A_P) = \emptyset$. □

♢ **Exemple 6.4**
*Soit* $\mathcal{F} = \{a^0, f^1, g^2\}$, *soit* $R = \{g(x,y) \to g(f(x), f(y))\}$ *un système de réécriture et soit* $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ *un automate tel que* $\mathcal{Q} = \{q_a, q_f\}$, $\mathcal{Q}_f = \{q_f\}$ *et* $\Delta = \{g(q_a, q_a) \to q_f, a \to q_a\}$.

*Soit* $\gamma$ *une fonction d'approximation tel que :*

$$Norm_\gamma(g(x,y) \to g(f(x), f(y)), q, \sigma) = \{g(q_1, q_2) \to q, f(\sigma(x)) \to q_1, f(\sigma(y)) \to q_2\}.$$

*On a* $R^*(A) = \{g(f^n(a), f^n(a)) \mid n \geq 0\}$.

*Après deux étapes de complétion, on obtient l'automate* $A_2 = (\mathcal{F}, \mathcal{Q}_2, \mathcal{Q}_f, \Delta_2)$ *avec* $\mathcal{Q}_2 = \mathcal{Q} \cup \{q_1, q_2\}$ *et* $\Delta_2 = \Delta \cup \{g(q_1, q_2) \to q_f, f(q_a) \to q_1, f(q_a) \to q_2, f(q_1) \to q_1, f(q_2) \to q_2\}$. *On a* $L(A_2) = \{g(f^*(a), f^*(a))\}$.

*Soit* $A_P$ *un automate reconnaissant uniquement le terme* $g(f(a), f(f(a)))$ *et* $\alpha$ *une fonction* $(A,R)-$*exacte. On a* $L(A_2) \cap L(A_P) \neq \emptyset$ *et* $L(A_2^e) \cap L(A_P) = \emptyset$, *les conditions* ***(1)*** *et* ***(2)*** *sont remplies. On peut en déduire qu'il existe un ensemble de transitions* $\Delta_0$ *qui soit discriminant. On peut prendre* $\Delta_0 = \{g(q1, q2) \to qf\}$, *il est discriminant contrairement à un* $\Delta'_0 = \{f(q1) \to q1\}$.

*On va maintenant raffiner la fonction* $\gamma$ *et on obtient :* $\gamma_{\Delta_0}(1) = q_{new1}$ *et* $\gamma_{\Delta_0}(2) = q_{new2}$.

*Il faut maintenant refaire la dernière étape de complétion avec la nouvelle fonction d'approximation, on calcule donc* $C^R_{\gamma_{\Delta_0}}(A_1)$. *On rajoute donc les transitions* $\{g(q_{new1}, q_{new2}) \to q_f, f(q_1) \to q_{new1}, f(q_2) \to q_{new2}\}$ *pour obtenir un nouvel automate* $A'_2$. *Et on a bien* $L(A'_2) \cap L(A_P) \neq \emptyset$.

*On réalise une autre étape de complétion à partir de $A'_2$, avec la fonction d'approximation $\gamma$ pour obtenir l'automate $A_3 = C^R_\gamma(A'_2) = (\mathcal{F}, \mathcal{Q}_3, \mathcal{Q}_f, \Delta_3)$ avec $\mathcal{Q}_3 = \mathcal{Q}_2$ et $\Delta_3 = \Delta_2 \cup \{f(q_{new1}) \to q_1, f(q_{new2}) \to q_2\}$.*

*L'automate point fixe est atteint, on a $L(A_3) \cap L(A_P) = \emptyset$ et on peut déduire que $R^*(L(A)) \cap L(A_P) = \emptyset$.*

Dans la section suivante, nous allons présenter l'analyse en arrière par complétion, qui, dans le cadre du raffinement d'approximation, servira à déterminer si un contre-exemple est un faux positif ou non. Si c'est un faux positif nous nous retrouvons dans le cas de la section 6.2.

## 6.3 Analyse en arrière par complétion

Dans le cas où sont réunies les conditions suivantes :

**(1')** $L(C^R_\gamma(A)) \cap L(A_P) \neq \emptyset$

**(2')** et $L(C^R_\alpha(A)) \cap L(A_P) \neq \emptyset$,

nous allons utiliser une technique d'analyse en arrière afin de se ramener dans le cas exposé en section 6.2 si possible. Nous sommes dans le cas d'un contre-exemple faux positif. Cette technique est schématisé par la figure 6.10. Notons que **(2')** implique **(1')**. Nous avons cependant écrit les deux inégalités par soucis d'analogie vis-à-vis des conditions de la section précédente.

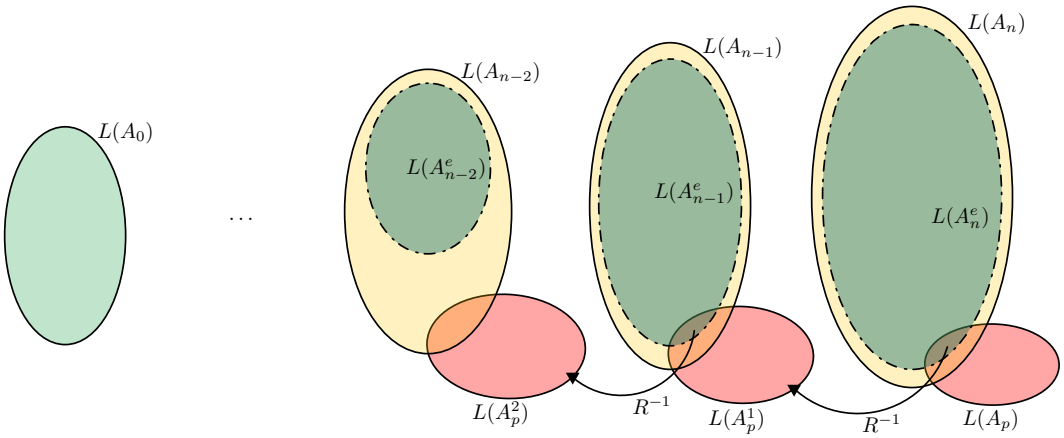



FIGURE 6.10 – Cas d'un contre-exemple faux positif

Cependant, il est possible que lors de l'analyse en arrière on ne retrouve pas les conditions de la section 6.2. Dans ce cas nous sommes en présence d'un vrai contre-exemple, voir la figure 6.11.

L'analyse en arrière par complétion peut être vue comme la complétion réalisée sur un automate $A$, en utilisant un système de réécriture $R$ dont le sens des règles a été inversé et une fonction d'approximation $(A, R)-$exacte. Soit $R^t$ le système de réécriture construit à partir de $R$ de la manière suivante : $R^t = \{r \to l \mid l \to r \in R\}$. Si $R$ satisfait la condition exprimant le fait que pour toutes règles de réécriture $l \to r \in R$ on a $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, alors il est possible que $R^t$ ne satisfasse pas cette condition (qui est requise pour les résultats sur la complétion). C'est pourquoi la définition de la complétion présentée en section 2.4 est étendue ici. Dans un premier temps, nous allons établir la relation entre $R^t(E)$ et $R^{-1}(E)$, avec $E \subseteq \mathcal{T}(\mathcal{F})$.

**♦ Proposition 6.5**
*Pour tout ensemble de termes $E \subseteq \mathcal{T}(\mathcal{F})$, on a $R^t(E) = R^{-1}(E)$.*

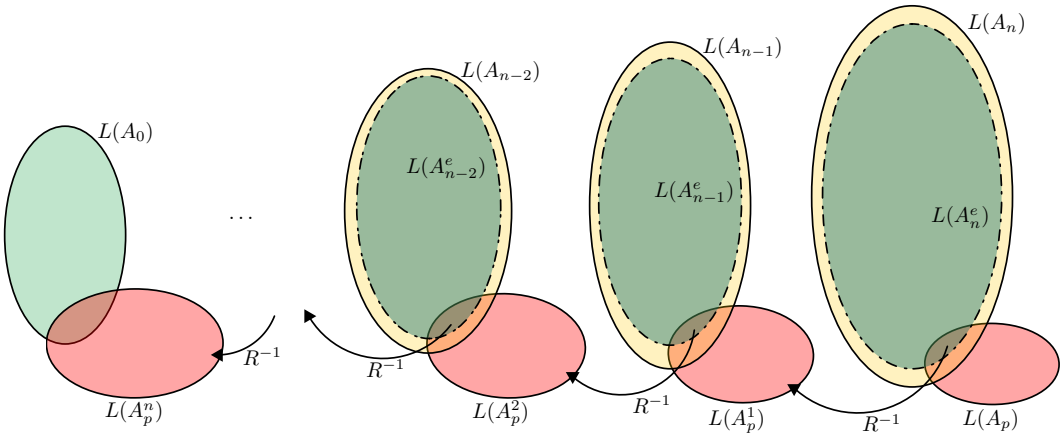



FIGURE 6.11 – Cas d'un vrai contre-exemple

*Démonstration.* La preuve est divisée en deux parties : **(a)** la preuve de $R^t(E) \subseteq R^{-1}(E)$, **(b)** la preuve de $R^t(E) \supseteq R^{-1}(E)$.

**(a)** Soient $t \in \mathcal{T}(\mathcal{F})$ et $t' \in E$ pour lesquels il existe $r \to l \in R^t$ tel que $t' \to_{\{r \to l\}} t$ et $t \in R^t(E)$. Si $r \to l \in R^t$ alors $l \to r \in R$ par définition de $R^t$. Alors on a $t \to_{\{l \to r\}} t'$ et $t \in R^{-1}(t')$. Par conséquent, pour chaque $t' \in E$, on a : $R^t(t') \subseteq R^{-1}(t')$.

**(b)** La preuve de $R^t(t') \supseteq R^{-1}(t')$ est similaire.

D'après **(a)** et **(b)**, on en déduit que $R^t(E) = R^{-1}(E)$. □

♢ **Exemple 6.6**
*Soit $R = \{g(x, f(y)) \to f(x)\}$. Le terme $f(a)$ se réécrit en $g(a, f(a))$ par $R^{-1}$. Il peut aussi se réécrire en $g(a, f(b))$ si $b \in \mathcal{F}$.*

Une conséquence directe de la proposition 6.5 est que $(R^t)^*(E) = (R^{-1})^*(E)$. Nous allons pouvoir maintenant définir l'algorithme de complétion permettant de faire des analyses sur $(R^{-1})^*(E)$.

Tout d'abord, afin de gérer les variables apparaissant en partie droite d'une règle de réécriture et non en partie gauche (comme la variable $y$ de l'exemple 6.6), on va définir un ensemble de transitions $T(q)$ en fonction d'un ensemble de symboles $\mathcal{F}$. Cet ensemble réduit chaque terme de $\mathcal{T}(\mathcal{F})$ en un état $q$. Il est constitué de toutes les transitions possibles utilisant uniquement l'état $q$.

△ **Définition 6.7**
*Soit $\mathcal{F}$ un ensemble de symboles, soit $q$ un état, $T(q)$ est l'ensemble de transitions :*

$$T(q) = \{s \to q \mid s \in T(\mathcal{F}), s = f(q, \ldots, q), f \in \mathcal{F}\}.$$

♢ **Exemple 6.8**
*Soit $\mathcal{F} = \{a^0, b^0, f^2\}$ un ensemble de symboles, soit $q_0 \in \mathcal{Q}$. L'ensemble de transitions $T(q)$ est :*

$$T(q_0) = \{a \to q_0,\ b \to q_0,\ f(q_0, q_0) \to q_0\}$$

À présent, on va définir une nouvelle substitution définie à partir d'un état, d'un ensemble de variables et d'une substitution initiale.

△ **Définition 6.9**
*Soit $q$ un état et $Y \subseteq \mathcal{X}$. Étant données une substitution $\sigma : \mathcal{X} \to \mathcal{Q}$ et une variable $x \in \mathcal{X}$, on pose :*

$$\mathrm{Chg}_Y^q(\sigma)(x) = \begin{cases} \sigma(x) & si\ x \in Y \\ q & sinon. \end{cases}$$

♢ **Exemple 6.10**
*Soit $\sigma$ une substitution telle que $\sigma(x) = q_1$ et $\sigma(y) = q_2$ et soit $q_0$ un état. On a :*

$$\mathrm{Chg}_{\{x\}}^{q_0}(\sigma)(x) = \sigma(x) = q_1,$$

$$\mathrm{Chg}_{\{x\}}^{q_0}(\sigma)(y) = q_0.$$

La définition suivante étend la définition 2.18 afin de gérer les systèmes de réécriture où il existe des règles de réécriture $l \to r$ où $\mathcal{V}ar(r) \not\subseteq \mathcal{V}ar(l)$.

△ **Définition 6.11** (Complétion en arrière)
*Soit $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ un automate d'arbres, $\gamma$ une fonction d'approximation et $q_{all}$ un état tel que $q_{all} \notin \mathcal{Q}$. Soit $R$ un système de réécriture linéaire à gauche et $\alpha$ une fonction d'approximation $(A,R)-$exacte. On définit un automate d'arbres $K_\alpha^R(A) = (\mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta')$ où :*

$$\Delta' = \Delta \cup \bigcup_{l \to r \in R, l\sigma \to_A^* q, r(\mathrm{Chg}_{\mathcal{V}ar(l)}^{q_{all}}(\sigma)) \not\to_A^* q} (Norm_\gamma(l \to r, \mathrm{Chg}_{\mathcal{V}ar(l)}^{q_{all}}(\sigma), q)) \cup T(q_{all}),$$

$$\mathcal{Q}' = \{q \mid c \to q \in \Delta'\}.$$

On peut voir que les variables des règles de réécritures apparaissant dans la partie droite et non dans la partie gauche sont traitées d'un façon particulière. En effet, chacune de ces variables est substituée par un état $q_{all}$ car il est impossible de déterminer quel terme se substitue à la place de ces variables. Grâce à l'ensemble de transitions $T(q_{all})$ tout terme $t \in \mathcal{T}(\mathcal{F})$ se réduit en un état $q_{all}$, $t \to_{T(q_{all})}^* q_{all}$. Par exemple, pour une règle de réécriture $f(x) \to g(x,y)$, la variable $y$ est substituée par l'état $q_{all}$, et on obtient $f(x) \to g(x, q_{all})$. Tout terme de la forme $f(x)$ est donc réécrit en terme de la forme $g(x,t)$ où $t$ est n'importe quel terme de $\mathcal{T}(\mathcal{F})$.

Enfin, en utilisant une fonction d'approximation $(A,R)-$exacte, on peut réaliser une analyse en arrière des termes atteignables grâce à la définition 6.11. Un automate construit à l'aide de cette définition à les propriétés énoncées dans la proposition 6.12 qui suit.

♦ **Proposition 6.12**
*Soit $A$ un automate d'arbres, $R$ un système de réécriture $\alpha$ fonction d'approximation. On a :*

**1)** *si $A$ est déterministe ou si $R$ est linéaire à gauche : $L(A) \cup R(L(A)) \subseteq L(K_\alpha^R(A))$;*

**2)** *si $R$ est linéaire et $\alpha$ est une fonction d'approximation $(A,R)-$exacte : $L(K_\alpha^R(A)) \subseteq R^*(L(A))$.*

La preuve de la proposition 6.12 est précédée par les deux lemmes 6.13 et 6.14 qui sont utiles à la preuve de la proposition.

▲ **Lemme 6.13**
*Soit $A$ un automate d'arbres, $R$ un système de réécriture et $\gamma$ une fonction d'approximation. Pour chaque substitution $\sigma$ de $\mathcal{X}$ dans $\mathcal{Q}$, si $l\sigma \to_A^* q$, alors $r\sigma \to_{K_\alpha^R(A)}^* q$.*

*Démonstration.* Si $r\sigma \to_A^* q$, alors $r\sigma \to_{K_\gamma^R(A)}^* q$ comme l'ensemble des transitions de $A$ est inclus dans l'ensemble des transitions de $K_\gamma^R(A)$.

Si $r\sigma \not\rightarrow_A^* q$, alors, par définition de $K_\gamma^R(A)$, $Norm_\gamma(l \to r, \sigma, q)$ est inclus dans l'ensemble des transitions de $K_\gamma^R(A)$. De plus, on peut facilement montrer que

$$r\sigma \rightarrow_{Norm_\gamma(l\to r,\sigma,q)}^* q,$$

ce qui prouve le lemme. □

Soit $A$ un automate d'arbres. Si un terme $t_1$ peut être réécrit en un terme $t_2$ en utilisant la transition $\delta$ à la position $p$, on écrit $t_1 \xrightarrow{p}_\delta t_2$. Formellement, $t_1 \xrightarrow{p}_\delta t_2$ si $t_{1|p}$ est la partie gauche de $\delta$ et si $t_2 = t_1[q]_p$, où $q$ est la partie droite de $\delta$. Il est évident que $t_1 \rightarrow_\delta t_2$ si et seulement si il existe une position $p$ de $t_1$ telle que $t_1 \xrightarrow{p}_\delta t_2$. L'ordre préfixe pour des mots $u$ et $v$ est défini comme suit : on dit que $u$ est un préfixe de $v$ s'il existe un mot $u'$ tel que $v = uu'$.

Le lemme suivant sera utile pour la preuve de la proposition 6.12.

**▲ Lemme 6.14**
*Soient $\delta_1$ et $\delta_2$ des transitions de l'automate d'arbres $A$, soient $t_1$, $t_2$ et $t_3$ des termes, et $p_1$ et $p_2$ des positions de, respectivement, $t_1$ et $t_2$ tel que $t_1 \xrightarrow{p_1}_{\delta_1} t_2 \xrightarrow{p_2}_{\delta_2} t_3$. Si $p_1$ et $p_2$ sont incomparables par ordre préfixe, alors il existe un terme $t_4$ tel que $t_1 \xrightarrow{p_2}_{\delta_2} t_4 \xrightarrow{p_1}_{\delta_1} t_3$.*

*Démonstration.* Soient $q_1$ et $q_2$ les parties droites des transitions $\delta_1$ et $\delta_2$ (respectivement). Si $t_1 \xrightarrow{p_1}_{\delta_1} t_2 \xrightarrow{p_2}_{\delta_2} t_3$, alors on a $t_2 = t_1[q_1]_{p_1}$, $t_3 = t_2[q_2]_{p_2}$ et $t_3 = (t_1[q_1]_{p_1})[q_2]_{p_2}$. Comme les positions $p_1$ et $p_2$ ne sont pas comparables par ordre préfixe alors on a aussi $t_3 = (t_1[q_2]_{p_2})[q_1]_{p_1}$.

Alors il existe un terme $t_4$ tel que $t_4 = t_1[q_2]_{p_2}$ et $t_3 = t_4[q_1]_{p_1}$. Ainsi on peut conclure que $t_1 \xrightarrow{p_2}_{\delta_2} t_4 \xrightarrow{p_1}_{\delta_1} t_3$ ce qui prouve le lemme. □

Nous allons maintenant prouver la proposition 6.12 en commençant par prouver **1)** puis **2)**.

*Démonstration.* de **1)**
Soit $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ un automate d'arbres, $R$ un système de réécriture et $\alpha$ une fonction d'approximation.

Comme $K_\alpha^R(A)$ est construit à partir de $A$ en y ajoutant des états et des transitions,

$$L(A) \subseteq L(K_\alpha^R(A)).$$

On suppose que $t \in R(L(A))$. Par définition il existe $t_0 \in L(A)$ tel que $t \in R(t_0)$. Alors il existe $l \to r \in R$, une position $p$ de $t_0$ et $\mu \in \Sigma(\mathcal{X}, \mathcal{T}(\mathcal{F}))$ tel que

$$t_{0|p} = l\mu \quad \text{et} \tag{6.1}$$
$$t = t_0[l\mu]_p. \tag{6.2}$$

De plus, comme $t_0 \in L(A)$ il existe $q \in Q$ et $q_f \in Q_f$ tels que

$$l\mu \rightarrow_A^* q \quad \text{and} \tag{6.3}$$
$$t \rightarrow_A^* t_0[q]_p \rightarrow_A^* q_f. \tag{6.4}$$

On a deux cas.
- Si $A$ est déterministe, alors pour chaque variable $x$ apparaissant dans $l$, (6.3) implique qu'il existe un unique état $q_x$ tel que $\mu(x) \rightarrow_A^* q_x$.
- Si $R$ linéaire à gauche, alors pour chaque variable $x$ apparaissant dans $l$, (6.3) implique qu'il existe un état $q_x$ tel que $\mu(x) \rightarrow_A^* q_x$.

La substitution $\sigma$ est définie par $\sigma(x) = q_x$ si $x$ apparaît dans $l$ et $\sigma(x) = q_{all}$ autrement. D'après (6.3) on a aussi :

$$l\mu \to_A^* l\sigma \to_A^* q \tag{6.5}$$

Il s'en suit que $r\mu \to_A^* r\sigma$. Ainsi, en utilisant le lemme 6.13, on a

$$r\mu \to_A^* r\sigma \to_{K_\alpha^R(A)}^* q \tag{6.6}$$

Donc, d'après (6.2), on a $t = t_0[r\mu]_p \to_{K_\alpha^R(A)}^* t_0[q]_p$. En utilisant (6.4), on a $t \to_{K_\alpha^R(A)}^* q_f$, ce qui prouve **1)**. □

*Démonstration.* de **2)**
Soit $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ un automate d'arbres, $R$ un système de réécriture linéaire à droite, et $\alpha$ une fonction d'approximation $(A, R)$-exacte.

On introduit une famille de langages $L_n \subseteq \mathcal{T}(\mathcal{F})$ définie par : $t \in L_n$ si et seulement si il existe des positions $p_1, \ldots, p_k$ de $t$, des transitions $\delta_1, \ldots, \delta_k$ de $K_\alpha^R(A)$, des termes $t_0, t_1, \ldots, t_k$ de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ tels que :

**i)** $t_0 \xrightarrow{p_1}_{\delta_1} t_1 \xrightarrow{p_2}_{\delta_2} t_2 \ldots \xrightarrow{p_k}_{\delta_k} t_k$, et

**ii)** $t_k \in \mathcal{Q}_f$ et $t_0 = t$, et

**iii)** $|\{i \mid \delta_i \notin \Delta \cup T(q_{all}\}| \leq n$.

Clairement, on a $L_0 = L(A)$, car les conditions expriment exactement l'existence d'une exécution réussie dans $A$.

Maintenant, on va prouver par induction sur $n$ la proposition suivante $\mathcal{P}_n$ : $L_n \subseteq R^*(L(A))$.

– Comme $L_0 = L(A)$, $\mathcal{P}_0$ est vrai.

– En supposant que $\mathcal{P}_n$ est vrai, soit $t \in L_{n+1}$. Par définition il existe des positions $p_1, \ldots, p_k$ de $t$, des transitions $\delta_1, \ldots, \delta_k$ de $K_\alpha^R(A)$, des termes $t_0, t_1, \ldots, t_k$ de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ tels que :

**i)** $t_0 \xrightarrow{p_1}_{\delta_1} t_1 \xrightarrow{p_2}_{\delta_2} t_2 \ldots \xrightarrow{p_k}_{\delta_k} t_k$, et

**ii)** $t_k \in \mathcal{Q}_f$ et $t_0 = t$, et

**iii)** $|\{i \mid \delta_i \notin \Delta \cup T\}| \leq n + 1$.

On note que si $p_i$ est un préfixe de $p_j$, alors $j \leq i$ (sachant que l'on travail sur des automates d'arbres bottom-up). Si $t \in L_0$ alors $t \in R^*(L(A))$. Ainsi on peut supposer, pour la suite de la preuve, que $t \notin L_0$. Par conséquent, il existe $\ell$ tel que $\ell = \min\{i \mid \delta_i \notin \Delta\}$. Comme $\delta_\ell \notin \Delta$, il existe $q \in \mathcal{Q}$ et $l \to r \in R$ tels que

$$l\sigma \to_A^* q \quad \text{etr}\sigma \not\to_A^* q \quad \text{et} \quad \delta_\ell \in Norm_\alpha(l \to r, \sigma, q).$$

Soit $p_m$ le préfixe maximal de $p_\ell$ tel que la partie droite de $\delta_m$ est dans $\mathcal{Q}$ (des transitions de $\delta_i$ peuvent avoir un partie droite dans l'ensemble des états de $K_\alpha^R(A)$ mais pas de $\mathcal{Q}$). Notons que comme on a $\{p_1, \ldots, p_k\} = \mathcal{P}os(t)$, $p_m$ est dans $\{p_1, \ldots, p_k\}$. Comme $t_k \in \mathcal{Q}_f \subseteq \mathcal{Q}$, $p_m$ existe.
En utilisant le lemme 6.14, on peut supposer, sans perdre de généralité, que la séquence de dérivation $i)$ satisfait : si $i \leq m$, alors $p_m$ est un préfixe de $p_i$. De plus, soit $P = \{p_i \mid i \leq m,\ \delta_i \notin \Delta\}$, et $P_{\max}$ les éléments de $P$ maximaux pour l'ordre préfixe. Par construction $p_\ell \in P_{\max}$. Soit $p_{\max} \in P_{\max}$. On suppose que l'ensemble $\{i \mid \ell < i < m,\ p_{\max}$ est un préfixe fixe de $p_i\}$ est non vide, soit $i_0$ son élément minimal. Pour tout $j \in \{\ell, \ell + 1, \ldots, i_0 - 1\}$, $p_j$ et $p_{i_0}$ sont incomparables suivant l'ordre du préfixe. En effet, comme $j < i_0$ si $p_j$ et $p_{i_0}$ sont comparables, alors $p_{i_0}$ est un préfixe de $p_j$. Dans ce cas,

$p_{\max}$ est un préfixe de $p_j$, ainsi $p_j \in \{i \mid \ell < i < m,\ p_{\max}$ est préfixe stricte de $p_i\}$, ce qui est une contradiction. Donc, en utilisant le lemme 6.14, on peut supposer sans perdre de généralité que la séquence de dérivation $i$) satisfait : si $i \leq m$, si $p_i$ a un préfixe dans $P_{\max}$, alors $i \leq \ell$.

En résumé, on a :

**i)** $t_0 \xrightarrow{p_1}_{\delta_1} t_1 \xrightarrow{p_2}_{\delta_2} t_2 \ldots \xrightarrow{p_k}_{\delta_k} t_k$, et

**ii)** $t_k \in \mathcal{Q}_f$ et $t_0 = t$, et

**iii)** $|\{i \mid \delta_i \notin \Delta \cup T\}| \leq n + 1$, et

**iv)** si $i \leq m$, alors $p_m$ est un préfixe de $p_i$, et

**v)** si $p_{\max} \in P_{\max}$ et si $p_{\max}$ est un préfixe de $p_i$ alors $i \leq \ell$.

On affirme maintenant que $t_m = t[q]_{p_m}$ et qu'il existe une substitution $\mu$ de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F})$ tel que $t_\ell = t[r\sigma]_{p_m}$. Soit $q_\ell$ la partie droite de $\delta_\ell$. Sachant que $\delta_\ell \in Norm_\alpha(l \to r, \sigma, q)$, alors soit $q_\ell = q$ et $m = \ell$, soit il existe $p'$ et $i$ tel que $q_\ell = \alpha(l \to r, \sigma, q)(p'.i)$. Comme $\alpha$ est une fonction d'approximation $(A, R)$-exacte, l'état $q_\ell$ apparaît dans une unique transition de $K^R_\alpha(A) : r(p')(\beta_{p.1}, \ldots, \beta_{p.n})$ où $\beta_{p.j}$ sont définies comme dans la définition 2.17. Inductivement en utilisant le fait qu'on a une fonction d'approximation $(A, R)$-exacte, on peut vérifier que $t_m = t[q]_{p_m}$ et que $t_\ell = t[r\sigma]_{p_m}$, ce qui prouve l'affirmation.

Enfin, sachant que $t_\ell = t[r\sigma]_{p_m}$, pour chaque variable $x$ apparaissant dans $r$, soit $\mu(x)$ un sous-terme de $t_{|p_m}$ tel que $\mu(x) \to^*_A \sigma(x)$ et $t = t[r\mu]_{p_m}$. Comme $R$ est linéaire à droite $\mu(x)$ est bien définie. Maintenant, pour chaque variable $x$ apparaissant dans $l$ mais non dans $r$, soit $\mu(x)$ un terme tel que $\mu(x) \to^*_A \sigma(x)$. le terme $t' = t[l\mu]_{p_m}$ satisfait $t \in R(t^p)$. De plus, $t' = t[l\mu]_{p_m} \to^*_A t[q]_{p_m} = t_m$. Par conséquent, $t' \in L_n$, ce qui prouve l'induction. On notera que $L(K^R_\alpha(A)) = \cup_{n \geq 0} L_n$, ce qui complète la preuve. □

Maintenant qu'on a défini l'analyse en arrière (dans cette section), ainsi que le raffinement de fonction d'approximation (section 6.2), on va définir un nouveau semi-algorithme pour la complétion d'automate d'arbres.

## 6.4 Semi-algorithme pour le raffinement

Dans la section 2.4, on a vu qu'on pouvait calculer une sur-approximation des termes atteignables, avec une fonction d'approximation correctement définie. Ceci nous permet de prouver (parfois) qu'un ensemble de termes n'est pas atteignable. De plus, avec un fonction d'approximation $(A, R)-$exacte, on peut calculer une sous-approximation des termes atteignables. Le semi-algorithme présenté en section 2.4 permet uniquement de faire ces deux analyses séparément.

Dans cette section, on va présenter un semi-algorithme qui fera ces deux types d'analyse à l'aide du raffinement d'approximation. Ce semi-algorithme nécessite en entrée les éléments suivants : $A$, $R$, $\gamma$, $\alpha$ et $A_p$ qui sont, respectivement, un automate d'arbre, un système de réécriture, une fonction d'approximation, une fonction d'approximation $(A, R)-$exacte et un automate reconnaissant des termes indésirables.

A l'aide de la complétion on calcule une séquence d'automates $A_0, \ldots, A_k$, où $A_0 = A$, jusqu'à ce que l'automate $A_k$ soit un point fixe, c'est-à-dire que $L(A_k) \cap L(A_p) = \emptyset$, ou jusqu'à ce qu'on ai $L(A_i) \cap L(A_p) \neq \emptyset$ $(0 \leq i \leq k)$. Dans le premier cas, à l'aide de la proposition 2.19, on peut conclure que les termes appartenant à $L(A_p)$ ne sont pas atteignables. Dans le deuxième cas, une

étape de complétion exacte est réalisée à partir de $A_{i-1}$ en utilisant $\alpha$. Si $L(C^R_\alpha(A_{i-1}))\cap L(A_p) = \emptyset$, alors on peut conclure que la fonction d'approximation $\gamma$ a été trop grossière (a fusionné trop d'états) à l'étape de complétion $i$. Alors, à l'aide de la proposition 6.3, une nouvelle fonction d'approximation $\gamma'$ est calculée de telle manière à ce qu'on ait $L(C^R_{\gamma'}(A_{i-1}))\cap L(A_p) = \emptyset$. Sinon, dans le cas où on a $L(C^R_\alpha(A_{i-1}))\cap L(A_p) \neq \emptyset$, l'analyse en arrière est utilisée afin de trouver une étape de complétion *coupable* $j$ $(0 < j < i)$, c'est-à-dire une étape pour laquelle on a $L(A_j)\cap L((K^R_\alpha)^{i-j}(A_p)) \neq \emptyset$. Une fois une étape de complétion coupable $j$ trouvée, on raffine la fonction d'approximation $\gamma$ et la complétion recommence à partir de l'étape $j$ avec la nouvelle fonction d'approximation raffinée (voir figure 6.10). Si aucune étape de complétion coupable n'est trouvée, alors on peut conclure $R^*(L(A))\cap L(A_p) \neq \emptyset$ (voir figure 6.11).

Dans l'algorithme 6.15, et jusqu'à la fin de cette section, on notera $A^e$ à la place de $C^R_\alpha(A)$, $A_i$ est le i-ième élément de la liste *aut_list*. Soit *list* une liste de $n$ éléments $[e_1, \ldots, e_n]$, on note $list[i]$ la sous liste $[e_1, \ldots, e_i]$, avec $i \leq n$. La fonction $list :: x$ ajoute l'élément $x$ à la fin de la liste *aut_list*.

De plus, dans l'algorithme 6.15 nous utilisons une fonction *Find* dont nous précisons le fonctionnement dans le paragraphe suivant l'algorithme.

**Algorithme 6.15** (Semi-algorithme pour le raffinement)
*Soient $R$ un système de réécriture linéaire, $A$ un automate d'arbres, $A_P$ un automate d'arbres reconnaissant un ensemble de termes non-désirés, $\gamma$ une fonction d'approximation et $\alpha$ une fonction d'approximation $(A,R)-$exact,* $\mathrm{Comp}_{\mathrm{Ref}}(A, R, A_P, \gamma, \alpha)$ *est défini comme suit :*

***Variables***
$A_P^{temp}$ := $A_P$ ;
*aut_list* := [$A$ ;$C^R_\gamma(A)$] ; (* liste d'automates *)
$i$ := 1 ; (* numéro d'étape de complétion *)
*result* := *true* ;
00 ***Begin***
01 ***While*** $(A_i \neq A_{i-1})$ ***and*** *(result = true)* ***do***
02 ***If*** $L(A_i)\cap L(A_P) = \emptyset$ ***then*** — *Si l'intersection est vide entre*
03 *aut_list* := *aut_list* :: $C^R_\gamma(A_i)$ ; — $L(A_i)$ *et* $L(A_P)$ *alors une étape*
04 $i := i + 1$ ; — *de complétion normale est réalisée.*
05 ***Else***
06 *status* := $L(C^R_\alpha(A_{i-1}))\cap L(A_P)$ ; — *Tant que l'intersection entre* $L(A^e_i)$
07 ***While*** *(status* $\neq \emptyset$*)* ***and*** $(i > 0)$ ***do*** — *et* $L(A_P)$ *est non vide, et tant que*
08 $A_P^{temp} := K^{R^t}_\alpha(A_P^{temp})$ ; — $i > 0$, *la définition 6.11 est utilisée*
09 $i := i - 1$ ; — *pour calculer une nouvel automate* $A_P$.
10 ***If*** $i > 0$ ***then***
11 *status* := $L(C^R_\alpha(A_{i-1}))\cap L(A_P^{temp})$ ;
12 ***EndIf***
13 ***Done*** — *Il y a deux cas où la boucle* **while** *s'arrête :*
14 $A_P^{temp}$ := $A_P$ ; — *- $i = 0$ (et $L(A^e_i)\cap L(A_P) \neq \emptyset$).*
15 ***If*** $(i = 0)$ ***then*** — *Dans ce cas on peut conclure que*
16 *result* := *false* ; — $R^*(L(A_0))\cap L(A_P) \neq \emptyset$ ;
17 ***Else*** — *- $L(A^e_i)\cap L(A_P) = \emptyset$ (et $i \geq 0$).*
18 *Find Disc* $\subseteq \Delta_i \setminus \Delta_{i-1}$ ;
19 $\gamma := \gamma_{Disc}$ — *Dans ce cas $\gamma$ est raffinée et*
20 *aut_list* := *aut_list[i-1]* :: $C^R_\gamma(A_i)$ ; — *une étape de complétion est réalisée.*
21 ***EndIf***
22 ***EndIf***
23 ***Done***
24 *return result* ;
25 ***End***

**Précisions sur la fonctionnement de la fonction *Find*.** Afin d'illustrer le fonctionnement de la fonction *Find*, on reprendra l'exemple de l'automate sur les mots de la section 6.1.1.

On a l'ensemble de symboles $\mathcal{F} = \{\omega^0, a^1, b^1, c^1, d^1, e^1, f^1\}$. On a l'automate $A_0 = (\mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_f, \Delta_0)$ (figure 6.2) reconnaissant le langage $L(A_0) = \{b(a(\omega))\}$, avec :

- $\mathcal{Q}_0 = \{q_1, q_2, q_3\}$,
- $\mathcal{Q}_f = \{q_3\}$,
- $\Delta_0 = \{\omega \to q_1, a(q_1) \to q_2, b(q_2) \to q_3\}$.

Après une étape de complétion on obtient l'automate $A_1 = (\mathcal{F}, \mathcal{Q}_1, \mathcal{Q}_f, \Delta_1)$ (figure 6.3), reconnaissant le langage $L(A_1) = \{f^*(\mathfrak{m}(d^*(\mathfrak{n}(\omega)))) \mid \mathfrak{m} \in \{b, e\}, \mathfrak{n} \in \{a, c\}\}$, avec :

- $\mathcal{Q}_1 = \mathcal{Q}_0$,
- $\Delta_1 = \Delta_0 \cup \{c(q_1) \to q_2, d(q_2) \to q_2, e(q_2) \to q_3, f(q_3) \to q_3\}$.

De plus on a l'automate $A_1^e = (\mathcal{F}, \mathcal{Q}_0^e, \mathcal{Q}_f, \Delta_0^e)$ (figure 6.4) reconnaissant la langage $L(A_1^e) = \{b(a(\omega)), b(d(c(\omega))), f(e(a(\omega))), f(e(d(c(\omega))))\}$.

Si on applique l'algorithme du raffinement d'approximation à cet exemple, la fonction *Find* s'éxécuterait à la première étape de complétion, lorsqu'on a

$$L(A_1) \cap L(A_p) \neq \emptyset \text{ et } L(A_1^e) \cap L(A_p) = \emptyset.$$

La fonction *Find* renvoie comme résultat un ensemble de transitions, dit discriminant, nommé *Disc*. Si on note $\Delta_1^+ = \Delta_1 \setminus \Delta_0$ l'ensemble de transitions ajouté à $\Delta_0$ pour obtenir $\Delta_1$, l'ensemble *Disc* est inclus dans $\Delta_1^+$.

Afin de déterminer quelles transitions de $\Delta_1^+$ sont dans *Disc*, chaque transition de $\Delta_1^+$ est ajoutée successivement à $\Delta_0$. On notera $A_1^t$ l'automate obtenu après chaque ajout. De plus, à chaque ajout, on teste l'intersection entre $L(A_1^t)$ et $L(A_p)$. On a donc deux cas :

- **1er cas** ($L(A_1^t) \cap L(A_p) \neq \emptyset$) **:** on ajoute à l'ensemble *Disc* la dernière transition ajoutée à $A_1^t$, et on la supprime des transitions de l'automate $A_1^t$, ensuite on ajoute une autre transition de $\Delta_1^+$ à $A_1^t$ ;
- **2e cas** ($L(A_1^t) \cap L(A_p) = \emptyset$) **:** on ajoute une autre transition de $\Delta_1^+$ à $A_1^t$.

Ainsi, après avoir ajouté les trois transitions $c(q_1) \to q_2$, $d(q_2) \to q_2$ et $e(q_2) \to q_3$ de $\Delta_1^+$ (transitions correspondantes aux transitions en tirets de la figure 6.12), on obtient un automate $A_1^t$ reconnaissant le langage $L(A_1^t) = \{\mathfrak{m}(d^*(\mathfrak{n}(\omega))) \mid \mathfrak{m} \in \{b, e\}, \mathfrak{n} \in \{a, c\}\}$ (figure 6.12). Et on a $L(A_1^t) \cap L(A_p) \neq \emptyset$ dès qu'on ajoute la transition $f(q_3) \to q_3$, donc on obtient $Disc = \{f(q_3) \to q_3\}$.

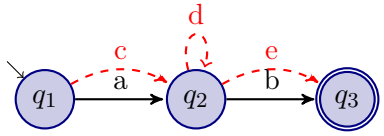



FIGURE 6.12 – Automate $A_1^t$

La transition de *Disc* étant issue de la normalisation de la transition $f(e(q2)) \to q3$ à l'aide de la fonction d'approximation $\gamma_1$, on va modifier cette dernière comme dans l'exemple de la section 6.1.1 afin d'obtenir l'automate de la figure 6.13 (qui ne reconnaît pas le terme indésirable).

Cependant, si on change l'ordre d'ajout des transitions, on peut obtenir un autre ensemble de transitions discriminant *Disc* qui permet d'obtenir par la suite un automate reconnaissant des termes indésirables. En effet, si après l'ajout de trois transitions, dont la transition détectée comme étant discriminante précédemment, on obtient l'automate $A_1^{t'} = (\mathcal{F}, \mathcal{Q}_1, \Delta_0 \cup \{d(q_2) \to$

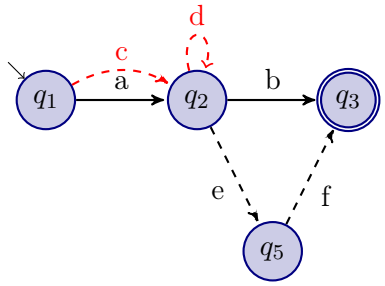



FIGURE 6.13 – Automate $A_1^{raff}$

$q_2, e(q_2) \to q_3, f(q_3) \to q_3\}$) (figure 6.14) tel que $L(A_1^{t'}) \cap L(A_p) = \emptyset$. Lorsqu'on ajoute la dernière transition $c(q_1) \to q_2$, on va une intersection non vide et $Disc = \{c(q_1) \to q_2\}$. On modifie en conséquence la fonction d'approximation, et on obtient :

$$\gamma'_{raff}(a(x) \to d(c(x)), \{x \to q_1\}, q_2)(1) = \{q_4\},$$

$$\gamma'_{raff}(b(x) \to f(e(x)), \{x \to q_2\}, q_3)(1) = \{q_3\}.$$

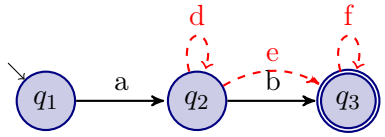



FIGURE 6.14 – Automate $A_1^{t'}$

Après une étape de complétion avec la nouvelle fonction d'approximation, on obtient l'automate $A_1^{raff'}$ (figure 6.15) reconnaissant le langage

$$L(A_1^{raff'}) = \{f^*(e(a(\omega)))), f^*(e(c(d(\omega))))), f^*(b(a(\omega)))), f^*(b(c(d(\omega))))), \}.$$

Et on peut voir qu'on a toujours une intersection non vide entre $L(A_1^{raff'})$ et $L(A_p)$.

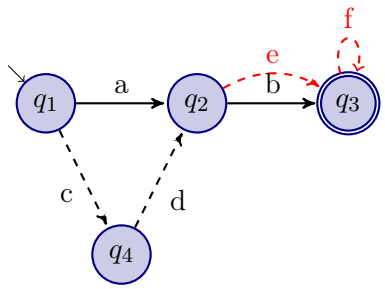



FIGURE 6.15 – Automate $A_1^{raff'}$

Afin de résoudre ce problème, on va ajouter successivement les transitions de $\Delta_1^+$, non pas aux transitions de l'automate $A_0$ mais aux transitions de l'automate $A_1^e$ (figure 6.4). Ainsi, on aura une intersection non vide entre $L(A_1^{t'})$ et $L(A_p)$ uniquement quand la transition $f(q_3) \to q_3$ appartient aux transitions de l'automate $A_1^{t'}$ (et que n'importe quelle(s) autre(s) transition(s) de $\Delta_1^+$ appartient aux transitions de $A_1^{t'}$).

Le théorème 6.16 énonce que ce semi-algorithme est correcte.

★ **Théorème 6.16** (Correction du semi-algorithme 6.15)
*Si* $\mathrm{Comp}_{\mathrm{Ref}}(A,\ R, A_P, \gamma, \alpha)$ *retourne true alors*

$$R^*(L(A)) \cap L(A_P) = \emptyset$$

*sinon, si* $\mathrm{Comp}_{\mathrm{Ref}}(A,\ R, A_P, \gamma, \alpha)$ *retourne false alors*

$$R^*(L(A)) \cap L(A_P) \neq \emptyset.$$

*Démonstration.* Le semi-algorithme 6.15 termine quand, soit un automate point fixe est calculé ou *result* = *false*. Si *result* = *false* alors la ligne 16 a été exécuté. De plus, d'après les lignes 7, 8, 9 et 11, $R^*(L(A)) \cap L(A_P) \neq \emptyset$. En effet, si $i = 0$ à la ligne 9 alors $i$ a eu comme valeur 1 soit à la ligne 6 (avant d'entrer dans la boucle **while**) ou soit à la ligne 9 (à l'itération précédente). Par conséquent, $L(C^R_\alpha(A)) \cap L(A_P^{temp}) \neq \emptyset$ comme $i$ prend ensuite la valeur 0, à la ligne 9. Par conséquent, comme $A_P^{temp}$ représente $(K_\alpha^{R^t})^k(A_P)$, d'après les propositions 6.12 and 6.5, on peut déduire que $R^*(L(A)) \cap L(A_P) \neq \emptyset$.

Si *result* = *true* et $A_i$ est un automate point fixe alors la ligne 15 n'a jamais été exécuté. Alors, pour que la boucle **while** de la ligne 7 s'arrête, *status* doit avoir comme valeur $\emptyset$. Alors, il existe $n \in \mathbb{N}$ et une fonction d'approximation $\gamma'$ construite à partir de $\gamma$ tel que $(C^R_{\gamma'})^{(n)}(A) = (C^R_{\gamma'})^{(n+1)}(A)$. D'après le théorème 2.20, on a $L((C^R_{\gamma'})^{(n)}(A)) \supseteq R^*(L(A))$. Par conséquent, on obtient $R^*(L(A)) \cap L(A_P) = \emptyset$.

Le cas où *result* = *false* et où $A_i$ est un automate point fixe, correspond au premier cas exposé dans cette preuve. □

Ensuite, le théorème 6.17 exprime le fait que le semi-algorithme est complet (partiellement) dans le sens où, si un terme non désiré est atteignable, alors le semi-algorithme 6.15 retourne false.

★ **Théorème 6.17** (Complétude partielle du semi-algorithme 6.15)
*Si* $R^*(L(A)) \cap L(A_P) \neq \emptyset$ *alors*

$$\mathrm{Comp}_{\mathrm{Ref}}(A, R, A_P, \gamma, \alpha) \text{ retourne } false.$$

*Démonstration.* On suppose qu'à chaque exécution, à la ligne 18, l'ensemble *Disc* est calculé tel que $Disc = \Delta_i \setminus \Delta_{i-1}$. Ce qui correspond à prendre une fonction d'approximation $\gamma$ identique à $\alpha$. Comme $R^*(L(A)) \cap L(A_P) \neq \emptyset$, il existe $n \in \mathbb{N}$ et $n > 0$ tels que $L((C^R_\gamma)^{(n)}(A)) \cap L(A_P) \neq \emptyset$ et $L((C^R_\gamma)^{(n-1)}(A)) \cap L(A_P) = \emptyset$. Par conséquent, dans cette configuration, *status* à la ligne 6 est différent de l'ensemble vide et $i = n$. Ensuite, $K^{R^t}_\alpha(A_P)$ est calculé. D'après les propositions 2.26 et 6.12, on peut déduire que $L((C^R_\gamma)^{(n-1)}(A)) \cap L(K^{R^t}_\alpha(A_P)) \neq \emptyset$. Alors, à l'aide d'un raisonnement par induction, on obtient $L((C^R_\gamma)^{(0)}(A)) \cap L((K^{R^t}_\alpha)^{(n)}(A_P)) \neq \emptyset$, qui correspond à la valeur stockée par *status*, à la ligne 11, après $n-1$ itérations de la boucle **while** de la ligne 7. Comme $i = 1$ et *status*$\neq \emptyset$, une nouvelle itération est réalisée. Enfin, $i = 0$ et *status*$\neq \emptyset$. Ainsi, *false* est affecté à *result*, la boucle **while** de la ligne 1 s'arrête et l'algorithme 6.15 retourne *false*. □

On a vu auparavant que la définition d'une fonction d'approximation nécessitait un opérateur humain ayant une bonne connaissance du système. Un avantage de l'approche par raffinement est que la fonction d'approximation $\gamma$ donnée en entrée de $\mathrm{Comp}_{\mathrm{Ref}}$ peut être quelconque, et donc automatique. En effet le semi-algorithme 6.15 permet de corriger automatiquement la fonction d'approximation afin d'obtenir une analyse conclusive si possible. Si l'analyse échoue, alors ce n'est pas dû à la fonction d'approximation mais au fait que le semi-algorithme indique que des termes non désirés sont atteignables. Ce semi-algorithme a été implémenté dans l'outils Timbuk [GVTT01] ainsi que dans l'outils TomedTimbuk [BBGM08].

## 6.5 Expérimentation : Système de deux processus à compteurs

Nous avons appliqué la technique de raffinement d'approximation pour la complétion d'automate d'arbres à la vérification d'un système de deux compteurs. Le système de réécriture de la figure 6.16 décrit le comportement de deux processus compteurs, chacun composé d'une liste d'entrée et d'une liste FIFO. Chaque processus reçoit une liste de symboles '+' et '−', qu'il va compter, en entrée. Un des deux processus, que l'on nommera $P_+$, compte les symboles '+' et l'autre processus, nommé $P_-$, compte les symboles '−'. Quand $P_+$ reçoit un '+', il le compte ; quand il reçoit un '−', il ajoute ce dernier à la liste FIFO de $P_-$. Le processus $P_-$ se comporte d'une manière symétrique. Quand la liste d'entrée et la liste FIFO d'un processus est vide, celui-ci s'arrête et renvoie la valeur de son compteur.

Le système complet des deux processus compteurs est modélisé de la façon suivante : le quadruplet `S(_,_,_,_)` représente le système complet ayant comme arguments le processus $P_+$, le processus $P_-$, la liste FIFO de $P_+$ et la liste FIFO de $P_-$. Le terme `Proc(_,_)` représente un processus où le premier argument est la liste d'entrée et le deuxième argument est le compteur. Le terme `add(_,_)` représente l'ajout d'un élément à une liste FIFO et les termes `cons`, `nil`, `s`, `o` sont les constructeurs habituels pour la représentation des nombres naturels.

**Symboles :** $\{nil^0, plus^0, o^0, end^0, minus^0, Stop^1, s^1, Proc^2, cons^2, add^2, S^4\}$
**Variables :** $\{x, y, z, u, c, m, n\}$
**Système de réécriture :**

$\{add(x, nil) \to cons(x, nil),$
$add(x, cons(y, z)) \to cons(y, add(x, z)),$
$S(Proc(cons(plus, y), c), z, m, n) \to S(Proc(y, s(c)), z, m, n),$
$S(Proc(cons(minus, y), c), u, m, n) \to S(Proc(y, c), u, m, add(minus, n)),$
$S(x, Proc(cons(minus, y), c), m, n) \to S(x, Proc(y, s(c)), m, n),$
$S(x, Proc(cons(plus, y), c), m, n) \to S(x, Proc(y, c), add(plus, m), n),$
$S(Proc(x, c), z, cons(plus, m), n) \to S(Proc(x, s(c)), z, m, n),$
$S(x, Proc(z, c), m, cons(minus, n)) \to S(x, Proc(z, s(c)), m, n),$
$S(Proc(nil, c), z, nil, n) \to S(Stop(c), z, nil, n),$
$S(x, Proc(nil, c), m, nil) \to S(x, Stop(c), m, nil)\}$

FIGURE 6.16 – Système de réécriture pour la système de deux compteurs

L'ensemble des configurations initiales est reconnu par l'automate d'arbres de la figure 6.17. Chaque processus a son compteur initialisé à 0 et leur liste d'entrée de taille non-bornée (comprenant des '+' et des '−') et avec au moins un symbole.

Avec cette spécification, on voudrait prouver que, pour toute liste d'entrée, aucun deadlock ne peut se produire. Dans notre cas, un deadlock est une configuration où un processus est arrêté alors qu'il reste toujours des symboles à compter dans sa liste FIFO. Cette propriété est spécifiée par un automate d'arbres `Bad_state` reconnaissant un système pour lequel un des deux processus s'est arrêté lorsque sa liste FIFO est non vide.

Avant de calculer une sur-approximation des configurations accessibles pour le système de deux processus, on va définir une fonction d'approximation $\gamma$ satisfaisant la propriété suivante : soient $\sigma_1$, et $\sigma_2$ deux substitutions de $\mathcal{X}$ dans $\mathcal{Q}$ telles que $\sigma_1 \neq \sigma_2$. Soit $l \to r$ une règle de réécriture de la figure 6.16, pour tout état $q$ et pour chaque position $p$ dans $r$, on a $\gamma(l \to r, \sigma_1, q)(p) = \gamma(l \to r, \sigma_2, q)(p)$.

**Etats :** $\{q_0, q_{init}, q_{zero}, q_{nil}, q_{list}, q_{symb}\}$
**Etat final :** $\{q_0\}$
**Transitions :**

$\{cons(q_{symb}, q_{nil}) \to q_{list}$, $cons(q_{symb}, q_{list}) \to q_{list}$, $o \to q_{zero}$,
$Proc(q_{list}, q_{zero}) \to q_{init}$, $S(q_{init}, q_{init}, q_{nil}, q_{nil}) \to q_0$, $nil \to q_{nil}\}$
$plus \to q_{symb}$, $minus \to q_{symb}$,

FIGURE 6.17 – Automate initial pour la système de deux compteurs

Avec cette fonction d'approximation, la complétion converge vers un point fixe $A$ après 8 étapes de complétion. Cependant, l'intersection entre l'automate $A$ et `Bad_state` est non vide. Comme $A$ est une sur-approximation, on ne peut donc pas savoir si oui ou non la spécification peut conduire à un deadlock.

Tandis qu'en utilisant la technique de raffinement d'approximation, il va être possible de conclure. En effet, cette fois-ci trois étapes de complétion vont être calculées et une intersection non-vide est trouvée. A partir de ce moment, une analyse en arrière est réalisée et qui, finalement, atteint l'automate d'arbres initial. On peut donc conclure que des deadlock peuvent se produire pour le système décrit par spécification.

Ce problème de deadlock peut être résolu en ajoutant un symbole `end` : celui-ci sera ajouté par le processus $P_+$ à la liste FIFO du processus $P_-$ quand $P_+$ a une liste d'entrée vide. On a le fonctionnement symétrique pour le processus $P_-$. Ainsi, un processus peut s'arrêter seulement si il a une liste d'entrée vide et si le symbole `end` est lu dans sa liste FIFO correspondante. Le système de réécriture de la figure 6.16 est modifié en conséquence.

Avec cette nouvelle spécification, et en utilisant le même type de fonction d'approximation décrite précédemment, la complétion d'automate d'arbres (sans raffinement d'approximation) calcule un automate point fixe $A'$ après 9 étapes de complétions. Et une fois de plus l'intersection entre $A'$ et `Bad_state` n'est pas vide. En utilisant le raffinement d'approximation, on obtient un automate point-fixe $A''$ dont le langage reconnu ne contient pas de terme reconnu par `Bad_state`. Par conséquent, on peut conclure que la nouvelle spécification ne produit pas de deadlock.

## 6.6 Bilan et perspectives

**Bilan** L'approche décrite dans cette section a pour objectif d'automatiser la méthode décrite à la section 2.4 et issue de [FGT04, Gen98]. Cette approche permet la génération automatique de sur-approximations, basées sur une fonction d'approximation et guidée par un ensemble de termes indésirables. Dans le cadre de la vérification de systèmes infinis, cette approche s'apparente à de l'abstraction guidée par une propriété de sûreté qui peut conclure que la propriété est vérifiée ou violée ou ne pas conclure. Ceci est dû au fait que le problème d'atteignabilité est indécidable pour les systèmes de réécriture non terminant.

Soit un automate initial $A_0$, un système de réécriture linéaire à gauche $R$, une fonction d'approximation $\gamma$, un automate $A_P$ représentant la propriété à vérifier et un automate $A_k$ tel que $L(A_k) \supseteq R^*(L(A_0))$. Auparavant, la technique exposée dans [FGT04, Gen98] permettaient de conclure uniquement qu'une propriété est vérifiée — $L(A_k) \cap L(A_P) = \emptyset$ —, ou alors ne permettaient pas de conclure — $L(A_k) \cap L(A_P) \neq \emptyset$ ou la procédure ne termine pas. Lorsque $L(A_k) \cap L(A_P) = \emptyset$, il fallait définir une nouvelle fonction d'approximation. Avec la technique de raffinement présentée, cette nouvelle fonction est définie automatiquement. Bien sûr la procédure

peut encore être non-conclusive ou ne pas terminer (le problème sous-jacent est indécidable).

Cependant, dans [BH08a], les auteurs montrent une limite des approches par approximation régulière pour l'analyse d'atteignabilité : une propriété vraie ne peut être prouvée avec aucune fonction d'approximation.

**Perspectives** Une première expérimentation (section 6.5) étant plutôt prometteuse, ainsi que les résultats obtenus dans le cadre de l'analyse de protocoles de sécurité et de programmes Java, nous permettent de penser que le raffinement d'approximation peut permettre à la technique détaillée dans [FGT04] d'être exploitable par une plus large communauté d'utilisateurs.

Plus d'expérimentations sont nécessaires afin de comparer la technique de raffinement et les abstractions présentées dans [BHRV05]. De plus il pourrait être intéressant de déterminer l'efficacité et la pertinence de l'analyse en arrière : à partir d'un système de réécriture $R$, d'un ensemble de termes initiaux $I$, et un ensemble de termes indésirables $P$, est-ce qu'il est plus intéressant de résoudre $R^*(I) \cap P = \emptyset$ ou $I \cap (R^{-1})^*(P) = \emptyset$ ? (sur le plan théorique, les problèmes sont duaux, mais en pratique, l'analyse en arrière est parfois plus efficace).

Lors du raffinement d'une fonction approximation on remarquera, qu'en pratique, il pourrait être intéressant d'essayer des heuristiques différentes pour le choix de l'ensemble des transitions discriminantes $Disc$. En effet, on peut voir qu'il est possible de prendre un ensemble $Disc$ égale à $\Delta_\gamma$ lorsque que l'on veut raffiner. Cependant, cette manière de raffiner pourrait s'avérer moins efficace qu'une recherche d'un ensemble $Disc$ ($\subseteq \Delta_\gamma$) le plus petit possible, car on risque d'introduire un trop grand nombre de nouvelles transitions et de nouveaux états qui feraient diverger la complétion.

De plus, il pourrait être intéressant de savoir si d'autres façons de raffiner la fonction d'approximation existent, tout en conservant la propriété énoncée en proposition 6.3. Par exemple, toujours dans un soucis de limiter la création de nouvelles transitions et de nouveaux états lors du raffinement, il pourrait être intéressant de réutiliser des états existants (et non de nouveaux états créés par la fonction d'approximation $(A, R)-$exacte).

# 7

# Cas des systèmes de réécriture non-linéaires à gauche

**Sommaire**



Dans ce chapitre, nous allons présenter une méthode étendant le domaine d'application de la technique de complétion décrite dans [FGT04] (section 2.4), précédemment restreinte aux systèmes de réécriture linéaires à gauche ou aux automates d'arbres déterministes.

Cette méthode (de complexité polynomiale) va permettre d'appliquer la complétion aux systèmes de réécriture non linéaires à gauche sans avoir d'opération de déterminisation à réaliser (opération de complexité exponentielle). Cette méthode peut s'apparenter à une déterminisation partielle d'un automate d'arbres, où les ensembles d'états nécessaires à une déterminisation classique sont limités par la taille.

La section 7.1 présente formellement la problématique et l'illustre avec un exemple. Dans la section 7.2, nous présentons deux domaines d'application de cette méthode : la vérification de protocole de sécurité et l'analyse en arrière de Bytecode Java. La section 7.3 contient la définition de la méthode illustrée par un exemple dans la section 7.4. La section 7.5 présente un bilan de cette méthode et les perspectives.

## 7.1 Problématique

Comme énoncé précédemment, la technique décrite en section 2.4, fonctionne uniquement quand le système de réécriture est linéaire à gauche ou aux automates d'arbres déterministes.

Par exemple, pour un automate d'arbres

$$A = (\{f, a\}, \{q_1, q_2, q_f\}, \{q_f\}, \{a \to q_1, a \to q_2, f(q_1, q_2) \to q_f\})$$

(figure 7.1), et un système de réécriture $R = \{f(x, x) \to g(x)\}$. La seule substitution $\sigma$ est telle que $\sigma(x) = q_1$ et $\sigma(x) = q_2$, ce qui est impossible. Comme il n'existe pas de substitution $\sigma$ telle que $l\sigma \to_A^* q$, avec $l = f(x, x)$, et pour $q$ un état de $\{q_1, q_2, q_f\}$, aucune transition n'est ajoutée. Cependant, comme on a $f(a, a) \in L(A)$, le terme $g(a)$ appartient à l'ensemble $R(L(A))$ (et non à l'ensemble $L(A)$). On voit sur cet exemple que la complétion s'arrête (ne commence pas en fait) avant de calculer une sur-approximation de $R^*(L(A))$.

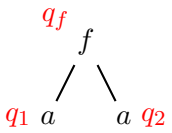



FIGURE 7.1 – Une exécution de $A$ sur $f(a, a)$.

Dans la section suivante nous présentons deux cas concrets où nous sommes confronté à ce problème. La section 7.3 montre comment modifier la procédure de complétion pour éviter ce problème et permettre de calculer, pour tout système de réécriture $R$ et pour tout automate d'arbres $A$, une sur-approximation de $R^*(L(A))$.

## 7.2 Présence de règles non-linéaires à gauche dans la pratique

### 7.2.1 Analyse de protocoles de sécurité

Les systèmes de réécriture utilisés dans le contexte de la vérification de protocoles de sécurité peut être parfois non linéaire à gauche. Cependant, si on veut réaliser une analyse par approximation, à l'aide de la technique décrite dans la section 2.4, on doit avoir un système de réécriture linéaire à gauche pour que l'analyse soit correcte. Néanmoins, cette technique peut être appliquée à des systèmes de réécriture non linéaires à gauche, si ils satisfont certaines conditions. Par exemple, dans [FGT04, GT01], les auteurs proposent d'assurer, lors de la complétion, le déterminisme des états qui correspondront aux variables non-linéaires. Ainsi, pour toute règle $l \to r$ non linéaire à gauche, pour chaque variables $x$ de $l$ non linéaires, et pour toute substitution $\sigma$ et pour tout état $q$ tel que $l\sigma \to_A^* q$, il existe un unique état $q'$ tel que $x\sigma \to_A^* q'$. Cependant, cette condition n'est pas adaptée pour être vérifiée automatiquement. En effet elle dépend de la définition de la fonction d'approximation, qui doit être modifiée si elle conduit à un automate ne vérifiant pas la condition ci-dessus.

Dans [BHK05], il est expliqué comment définir certains critères sur $R$ (non linéaire à gauche) et $A$, pour que la complétion soit correcte pour l'analyse de protocoles industriels. Ces critères sont basés sur un typage des termes et sur le fait que seulement les termes d'un certain type peuvent se substituer à des variables non-linéaires à gauche.

Dans [BHK06], les auteurs définissent une procédure de complétion modifiée afin de gérer automatiquement l'analyse de systèmes ayant des propriétés algébriques, comme le `XOR` par exemple. Les règles de réécriture $l \to r$ non linéaire à gauche ont leur variables (non linéaires) renommées en $x_i$. Pour une substitution $\sigma$ telle que $l\sigma \to_A^* q$, la procédure ajoute la transition $r\sigma \to q$ si, pour tout $x_i \in \mathcal{V}ar(l)$, $\bigcap_{i\in\mathbb{N}} L(A, \sigma(x_i)) \neq \emptyset$.

Les critères définis dans [FGT04] ne permettent pas de gérer les règles de réécriture non linéaire à gauche du `XOR`, par exemple. Dans [BHK05], les substitutions de variables de règles non linéaires à gauche sont limitées suivant les types de termes. Cependant, certaines attaques de protocoles sont provoquées par des confusions sur les types [CDL06, BLP03, CDSV05]. L'approche de [BHK06] peut s'appliquer à n'importe quel système de réécriture. Notamment avec les systèmes de réécriture comprenant des règles de gestion des propriétés algébriques de l'exponentiel, ce qui permet l'analyse de protocoles basés sur l'échange de clé Diffie-Hellman.

### 7.2.2 Analyse en arrière de Bytecode Java

Un récent travail [BGJR07], dédié à l'analyse statique de bytecode Java à l'aide de système de réécriture, fournit une procédure automatique de traduction du bytecode d'un programme Java

et du fonctionnement de la machine virtuelle Java en un système de réécriture. Ce système de réécriture généré est linéaire à gauche et quadratique à droite. Par exemple, la règle de réécriture suivante :

$$xInvokeSpecialCC(pprotected, val_{true}, cc, ca, cam, ic) \rightarrow xInvokeSpecialCC(subclass(ic, cc), val_{true}, cc, ca, cam, ic)$$

n'est pas linéaire à droite, où $xInvokeSpecial^2$, $pprotected^0$ et $val^0_{true}$ sont des symboles et $ca$, $ic$, $cam$ et $cc$ sont des variables. Cette règle constitue une partie de la traduction du bytecode Java concernant l'invocation de méthodes.

Si on veut appliquer le raffinement d'approximation (chapitre 6), ou uniquement effectuer une analyse en arrière (section 6.3), le sens des règles du système de réécriture doit être inversé. On obtient donc un système de réécriture quadratique à gauche et linéaire à droite, ce qui ne permet pas d'effectuer de raffinement d'approximation ou d'analyse en arrière à moins de déterminiser les automates complétés par le système de réécriture quadratique.

## 7.3 Calcul de $A^{(k)}$

Dans cette section nous allons étendre la technique introduite dans [FGT04] aux systèmes de réécriture non linéaires à gauche. Dans le cas où le système de réécriture n'est pas linéaire à gauche alors il est possible d'appliquer la complétion (section 2.4) en déterminisant les automates calculés. Cependant cette déterminisation est de complexité exponentielle et ce calcul doit être réalisé à chaque étape de complétion car elle ne préserve pas le déterminisme de l'automate. Cette méthode est alors trop coûteuse en temps de calcul pour être utilisée en pratique.

La méthode proposée par la suite permet d'éviter les calculs de complexité exponentielle. Celle-ci fournit un modèle paramétrable et un outil théorique pour les systèmes de réécriture $h$-linéaires à gauche, pour un $h$ petit. La petite valeur de $h$ peut être justifiée par le fait que la plupart des systèmes de réécriture, modélisant des systèmes concrets, sont au plus 2-linéaires à gauche. En effet, les règles de réécriture 2-linéaire à gauche modélisent des comparaisons entre des éléments du système analysé. Et dans la plupart des cas pratiques, seulement deux éléments sont comparés.

Nous sommes donc dans la situation suivante, où nous avons les données suivantes :
- un automate d'arbres (non déterministe) $A$,
- un système de réécriture $R$ $h$-linéaire à gauche.

On ne peut donc pas appliquer la méthode de complétion pour cet automate et ce système de réécriture.

Comme on a pu le voir dans la section 7.1, aucune étape de complétion est réalisée car il n'existe pas de substitution valide. Bien que par exemple, pour une substitution $\sigma(x)$, où $x$ est une variable non linéaire, pour deux états $q$ et $q'$ de $A$ tels que $\sigma(x) = q = q'$, on ait l'égalité entre les langages $L(A, q) = L(A, q')$.

L'idée, définie formellement dans la définition 7.1, va être de calculer un automate $A^{(h)}$ à partir de $A$, reconnaissant tous deux le même langage tel que, si on reprend l'exemple ci-dessus, on aura $L(A, q) = L(A, q') = L(A^{(h)}, \{q, q'\})$, où $\{q, q'\}$ est un état de $A^{(h)}$.

△ **Définition 7.1**
*Soit $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ un automate d'arbres. L'automate $A^{(h)} = (\mathcal{F}, \mathcal{Q}^{(h)}, \mathcal{Q}_f^{(h)}, \Delta^{(h)})$, pour $h \geq 1$, est défini par :*
- $\mathcal{Q}^{(h)} = \{\{q\} \mid q \in \mathcal{Q}\} \cup \{Q \subseteq \mathcal{Q} \mid Card(Q) \leq h\}$ *(les états de $\mathcal{Q}^{(h)}$ sont définis à l'aide de l'exposant $(h)$),*
- $\mathcal{Q}_f^{(h)} = \{\{q\} \mid q \in \mathcal{Q}_f\}$,

– $\Delta^{(h)} = \{f(q_1^{(h)}, \ldots, q_n^{(h)}) \to q^{(h)} \mid \forall q \in q^{(h)}, \exists q_1, \ldots, q_n \in \mathcal{Q}, \forall 1 \leq i \leq n, q_i \in q_i^{(h)} \text{ et } f(q_1, \ldots, q_n) \to q \in \Delta\}$.

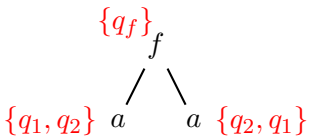



FIGURE 7.2 – Une exécution de $A^{(2)}$ sur $f(a, a)$.

Pour illustrer la définition précédente, considérons l'automate d'arbres $A$ ayant pour état final $q_f$ et dont les transitions sont $a \to q_1$, $A \to q_2$ et $f(q_1, q_2) \to q_f$. Les états de $A^{(2)}$ sont tous les paires d'états et tous les singletons faisables à partir de l'ensemble $\{q_1, q_2, q_f\}$. Les transitions de $A^{(2)}$ sont $A \to \{q_1\}$, $a \to \{q_2\}$, $a \to \{q_1, q_2\}$, $f(\{q_1\}, \{q_2\}) \to \{q_f\}$, $f(\{q_1, q_i\}, \{q_2, q_j\}) \to \{q_f\}$ pour tous les $i, j$ dans $\{1, 2, f\}$ (figure 7.2). En considérant uniquement les états accessibles, parmi toutes les transitions précédentes on obtient uniquement les transitions $f(\{q_1, q_i\}, \{q_2, q_j\}) \to \{q_f\}$ pour tous les $i, j$ dans $\{1, 2\}$.

De plus, on a l'égalité entre les langages reconnus par $A$ et $A^{(h)}$.

**♦ Proposition 7.2**
*Pour tout automate d'arbres $A$, on a $L(A) = L(A^{(h)})$.*

*Démonstration.* Par définition de $A^{(h)}$, si $f(q_1, \ldots, q_n) \to q \in \Delta$, alors $f(\{q_1\}, \ldots, \{q_n\}) \to \{q\} \in \Delta^{(h)}$. Par conséquent, pour chaque terme $t$ tel que $t \to_A^* q$, on a aussi $t \to_{A^{(h)}}^* \{q\}$, car pour tout $q_f \in \mathcal{Q}_f$, $\{q_f\} \in \mathcal{Q}_f^{(h)}$, $L(A) \subseteq L(A^{(h)})$.

Il reste à prouver que $L(A^{(h)}) \subseteq L(A)$. Nous allons prouver par induction sur $k$ que pour chaque $k \geq 1$, pour chaque terme $t$, chaque état $q^{(h)}$ de $A^{(h)}$, si $t \to_{A^{(h)}}^k q^{(h)}$, alors pour tout $q \in q^{(h)}$, $t \to_A^k q$.

– Si $t \to_{A^{(h)}} q^{(h)}$, alors, par définition de $\Delta^{(h)}$, $t$ est d'arité 0 et pour tout $q \in q^{(h)}$, il existe une transition $t \to q$ dans $A$.
– On considère maintenant que l'hypothèse est vraie pour un entier positif fixé $k$. Soit $t$ un terme et $q^{(h)} \in A^{(h)}$ tel que $t \to_{A^{(h)}}^{k+1} q^{(h)}$. Par conséquent, il existe $f \in \mathcal{F}_n$ tel que

$$t \to_{A^{(h)}}^k f(q_1^{(h)}, \ldots, q_n^{(h)}) \to_{A^{(h)}} q^{(h)}.$$

Il s'en suit $t = f(t_1, \ldots, t_k)$ et pour tout $1 \leq i \leq k$, $t_i \to_{A^{(h)}}^k q_i^{(h)}$. En utilisant l'hypothèse d'induction, $t_i \to_A^k q_i$, pour tout $q_i \in q_i^{(h)}$. Par conséquent, pour tout $q \in q^{(h)}$, $f(q_1, \ldots, q_n) \to q \in \Delta$, ce qui prouve l'induction.

Il en résulte que $L(A^{(h)}) \subseteq L(A)$. □

Nous allons maintenant montrer qu'une étape de complétion réalisée sur un automate $A^{(h)}$, avec un fonction d'approximation quelconque est une sur-approximation de l'ensemble de terme $R(L(A)) \cup L(A)$. On montrera aussi que pour une fonction d'approximation $(A, R)$-exacte, la complétion de $A^{(h)}$ calcul une sous-approximation de $R^*(L(A))$.

Nous allons tout d'abord montrer que si un terme $t \in \mathcal{T}(\mathcal{F} \cup \{q_1, \ldots, q_n\})$, où $q_1, \ldots, q_n$ sont des états, se réduit en un état $q$ par les transitions d'un automate $A$, alors le terme $t \in \mathcal{T}(\mathcal{F} \cup \{q_1^{(h)}, \ldots, q_n^{(h)}\})$ se réduit en un état $\{q\}$ par des transitions de l'automate $A^{(h)}$ si pour tout $1 \leq i \leq n$ on a $q_i \in q_i^{(h)}$.

**▲ Lemme 7.3**
*Soit $k \geq 1$. Si $C[q_1, \ldots, q_n] \to_A^k q$ et si $q_1^{(h)}, \ldots q_n^{(h)}$ sont des états de $A^{(h)}$ satisfaisant $q_i \in q_i^{(h)}$ pour tout $1 \leq i \leq n$, alors $C[q_1^{(h)}, \ldots, q_n^{(h)}] \to_{A^{(h)}}^k \{q\}$.*

*Démonstration.* On va prouver par induction sur $k$ que pour chaque $k \geq 1$, si $C[q_1, \ldots, q_n] \to_A^k q$ et si $q_1^{(h)}, \ldots q_n^{(h)}$ sont des états de $A^{(h)}$ satisfaisant $q_i \in q_i^{(h)}$ pour tout $1 \leq i \leq n$, alors $C[q_1^{(h)}, \ldots, q_n^{(h)}] \to_{A^{(h)}}^k \{q\}$.

- Si $k = 1$, alors $C[q_1, \ldots, q_n] \to q$ est une transition de $A$. Alors, par définition de $\Delta^{(h)}$, $C[q_1^{(h)}, \ldots, q_n^{(h)}] \to \{q\}$ est une transition de $A^{(h)}$.
- On considère que la proposition est vraie pour tout $j \leq k$ et que $C[q_1, \ldots, q_n] \to_A^{k+1} q$. Il existe alors $q'_1, \ldots, q'_\ell$ des états de $A$ et $f \in \mathcal{F}_\ell$ tels que $C[q_1, \ldots, q_n] \to_A^k f(q'_1, \ldots, q'_\ell) \to_A q$. Par conséquent, $C[q_1, \ldots, q_n]$ est de la forme

  $$C[q_1, \ldots, q_n] = f(t_1, \ldots, t_\ell)$$

  où les $t_i$ sont des termes sur l'alphabet $\mathcal{F} \cup \{q_1, \ldots, q_n\}$. De plus, pour tout $i$, il existe $k_i \leq k$ tel que $t_i \to_A^{k_i} \{q'_i\}$ et $\sum_i k_i = k$. Alors, à l'aide de l'hypothèse d'induction, on sait que

  $$t_i^{(h)} \to_{A^{(h)}}^{k_i} \{q'_i\}$$

  où $t_i^{(h)}$ est le terme obtenu à partir de $t_i$ par substitution de $q_i$ par $q_i^{(h)}$. Maintenant, sachant que $f(q'_1, \ldots, q'_\ell) \to q$ est une transition de $A$, $f(\{q'_1\}, \ldots, \{q'_\ell\}) \to \{q\}$ est alors une transition de $A^{(h)}$.
  On peut conclure que $C[q_1^{(h)}, \ldots, q_n^{(h)}] \to_{A^{(h)}}^{k+1} \{q\}$, ce qui prouve le lemme.

□

**▲ Lemme 7.4**
*S'il y a $q_1, q_2, \ldots, q_j$ des états de $A$, avec $j \leq h$ tel que $t \to_A^* q_i$ pour tout $1 \leq i \leq j$, alors $t \to_{A^{(h)}}^* \{q_i \mid 1 \leq i \leq j\}$.*

*Démonstration.* Si $t \to_A^* q_i$ pour chaque $1 \leq i \leq j$, alors il existe des fonctions $\pi_i$ d'une position de $t$ dans $\mathcal{Q}$ tel que $\pi_i(\varepsilon) = q_i$ et pour chaque position $p$ de $t$, si $t|_p \in \mathcal{F}_n$, alors $t(p)(\pi_i(p.1), \ldots, \pi_i(p.n)) \to \pi_i(p)$ est une transition de $A$.

Alors, par définition de $\Delta^{(h)}$, $t(p)(\{\pi_i(p.1) \mid 1 \leq i \leq j\}, \ldots, \{\pi_i(p.n) \mid 1 \leq i \leq j\}) \to \{\pi_i(p) \mid 1 \leq i \leq j\}$ est dans $\Delta^{(h)}$. il s'en suit que $t \to_{A^{(h)}}^* \{q_i \mid 1 \leq i \leq j\}$. □

**♦ Proposition 7.5**
*Si chaque variable apparaît au plus h fois dans la partie gauche de la règle de R, alors pour toute fonction d'approximation $\gamma$, on a $R(L(A)) \cup L(A) \subseteq L(\mathcal{C}_\gamma^R(A^{(h)}))$.*

*Démonstration.* Étant donné que $L(A) = L(A^{(h)})$ et que $L(A^{(h)}) \subseteq L(\mathcal{C}_\gamma^R(A^{(h)}))$, on a $L(A) \subseteq L(\mathcal{C}_\gamma^R(A^{(h)}))$. Il reste à montrer que $R(L(A)) \subseteq L(\mathcal{C}_\gamma^R(A^{(h)}))$.

Soit $t \in R(L(A))$. Par définition il existe une règle $l \to r \in R$, une position $p$ de $t$ et une substitution $\mu$ de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F})$ tel que

$$t = t[r\mu]_p \quad \text{et} \quad t[l\mu]_p \in L(A) \tag{7.1}$$

Il en résulte qu'il existe des états $q, q_f$ de $A$ tel que $q_f$ est final, et

$$l\mu \to_A^* q \quad \text{et} \quad t[q]_p \to_A^* q_f. \tag{7.2}$$

Par conséquent, à l'aide de la définition 7.1, on a

$$l\mu \to^*_{A^{(h)}} \{q\} \quad \text{et} \quad t[\{q\}]_p \to^*_{A^{(h)}} \{q_f\}. \tag{7.3}$$

Si $r\mu \to^*_{A^{(h)}} \{q\}$, alors (7.3) implique que $t[r\mu]_p \to^*_{A^{(h)}} \{q_f\}$. Dans ce cas, comme $t = t[r\mu]_p$ et comme $\{q_f\}$ est, par construction, un état final de $A^{(h)}$, $t$ appartient à $L(A^{(h)})$, ce qui est un sous-ensemble de $L(\mathcal{C}_\gamma(A^{(h)}))$.

On considère maintenant que $r\mu \not\to^*_{A^{(h)}} \{q\}$. Soit $P_l$ l'ensemble des positions des variables de $l$ défini par $P_l = \{p \mid l(p) \in \mathcal{X})\}$. Posons $P_l = \{p_1, \ldots, p_\ell\}$. Comme $l\mu \to^*_A q$, d'après (7.2) il existe des états $q_1, \ldots, q_\ell$ de $A$ tel que

$$\mu(l(p_i)) \to^*_A q_i \quad \text{et} \quad l[q_1]_{p_1} \ldots [q_\ell]_{p_\ell} \to^*_A q. \tag{7.4}$$

On définit la substitution $\sigma$ des variables apparaissant dans $l$ dans $2^{\mathcal{Q}}$ par : $\sigma(x_i) = \{q_i \mid l(p_i) = x_i\}$. Comme $l$ est h-linéaire à gauche, pour chaque $x_i$, $\sigma(x_i)$ contient au plus h états. On affirme que $l\sigma \to^*_{A^{(h)}} q$. En effet, par (7.4) et d'après le lemme 7.4, pour chaque $x_i$ apparaissant dans $l$, $\mu(x_i) \to^*_{A^{(h)}} \sigma(x_i)$. On obtient $l\mu \to^*_{A^{(h)}} l\sigma$. D'après (7.4) et en utilisant le lemme 7.3, $l\sigma \to^*_{A^{(h)}} \{q\}$, ce qui prouve l'affirmation. Par construction de $\mathcal{C}^R_\gamma(A^{(h)})$, $r\sigma \to^*_{\mathcal{C}^R_\gamma(A^{(h)})} \{q\}$. De plus, par définition de $\sigma$, $r\mu \to^*_{A^{(h)}} r\sigma$. On obtient que

$$t = t[r\mu]_p \to^*_{A^{(h)}} t[r\sigma]_p \to^*_{\mathcal{C}_\gamma(A^{(h)})} t[\{q\}]_p \to^*_{A^{(h)}} \{q_f\},$$

ce qui termine la preuve. □

♦ **Proposition 7.6**
*Si $R$ est linéaire à droite et si $\alpha$ est une fonction d'approximation $(A, R)$-exacte, alors*

$$L(\mathcal{C}^R_\alpha(A^{(h)})) \subseteq R^*(L(A)).$$

*Démonstration.* C'est une conséquence directe du théorème 2.26 et de la proposition 7.2. □

Notons que si $A$ est un automate d'arbres non-déterministe pour $n$ états, tout automate déterministe reconnaissant $L(A)$ aura $O(2^n)$ états. De plus, en considérant $h$ comme une constante, l'automate $A^{(h)}$ a $O(n^h)$ états et $O(n^{hk})$ transitions, où $k$ est l'arité maximale des symboles dans $\mathcal{F}$. Pour un système de réécriture 2-linéaire à gauche, $A^{(h)}$ a, au maximum, significativement moins d'états que l'automate déterministe reconnaissant $L(A)$.

## 7.4 Exemple

Nous avons illustré notre approche sur la famille d'exemples suivant. Nous considérons tout d'abord la famille d'automates d'arbres $(A_n)$ définie par : l'ensemble des états de $A_n$ est $\{q_1, \ldots, q_{2n+2}, q_f\}$, l'ensemble de ses états finaux est $\{q_f\}$, et l'ensemble des ses transitions est, pour tout $i \geq 3$ :

$\{\omega \to q_1, \quad a(q_1) \to q_1, \quad b(q_1) \to q_1, \quad a(q_1) \to q_3, \quad a(q_i) \to q_{i+2},$
$\omega \to q_2, \quad a(q_2) \to q_2, \quad b(q_2) \to q_2, \quad a(q_2) \to q_4, \quad b(q_i) \to q_{i+2}, \quad f(q_{2n+1}, q_{2n+2}) \to q_f\}.$

L'automate $A_n$ (figure 7.3) accepte l'ensemble de termes de la forme $f(t_1, t_2)$ où $t_1$ et $t_2$ sont des termes de $T(\{a^1, b^1, \omega^0\})$ tel que $t_1|_{1^{n-1}}$ et $t_2|_{1^{n-1}}$ existent et sont dans $\{a\}.\{a, b\}^*$.

Informellement, on ramène un terme de la forme $a(b(\omega))$ au mot $ab$, et chaque termes $t_1$ et $t_2$ peuvent être vue comme des mots de $\{a, b\}^{n-1}.\{a\}.\{a, b\}^*$ satisfaisant la condition précédemment énoncée.

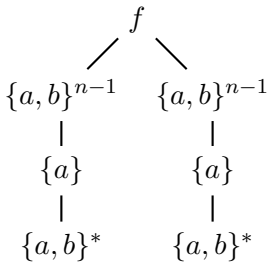



FIGURE 7.3 – Représentation de l'automate $A_n$.

On considère le système de réécriture $R$ contenant uniquement la règle de réécriture $f(x, x) \to x$. On veut prouver que $b^{n-1}a(\omega) \in R^*(L(A_n))$. En utilisant la méthode de complétion directement, on ne pourra pas prouver ce résultat. En déterminisant l'automate $A_n$, ainsi que chaque automate issu des étapes de complétion, on peut prouver ce résultat. En effet la seule substitution est telle que $\sigma(x) = q_{2n+1}$ et $\sigma(x) = q_{2n+2}$ alors, comme il n'existe pas de substitution $\sigma$ telle que$f(\sigma(x), \sigma(x)) \to^*_{A_n} q$ aucune étape de complétion n'est réalisée. Cependant, l'automate minimal de $L(A_n)$ a $2^n$ états au moins. Ensuite la complétion doit être exécuté sur cet automate ce qui se traduit par un temps d'exécution exponentiel pour prouver que $b^{n-1}a(\omega) \in R^*(L(A_n))$. En utilisant l'approche décrite précédemment en section 7.3, on peut calculer un automate $A_n^{(h)}$, et en appliquant la complétion sur cet automate, on pourra prouver le résultat en temps polynomial.

## 7.5 Bilan et perspectives

Dans ce chapitre nous avons proposé une extension de la procédure de complétion aux systèmes de réécriture non linéaires à gauche.

Initialement, dans le cas d'un système de réécriture non linéaire à gauche, la procédure de complétion devait s'effectuer en utilisant un automate déterministe. Cependant, chaque étape de complétion ne préservant pas le déterminisme de l'automate, une phase de déterminisation de l'automate s'effectuant en un temps exponentiel est alors nécessaire. En pratique cette solution ne convient pas car elle conduit à des automates de grandes tailles ne permettant pas la bonne exécution de la procédure de complétion.

Nous avons présenté une nouvelle procédure s'effectuant en un temps polynomial permettant de calculer un automate $A^{(h)}$ et remplaçant la phase de déterminisation nécessaire lors de la complétion d'un automate par un système de réécriture non linéaire à gauche. Cette approche est le plus efficace lorsque que le système de réécriture est quadratique à gauche, c'est-à-dire quand $h$ égale 2. Dans la pratique on a vue que les systèmes de réécriture quadratiques à gauche sont présents pour la vérification de protocoles de sécurité et de programmes Java.

Cependant, bien que cette nouvelle procédure améliore théoriquement les temps de calcul par rapport à une déterminisation, dans la pratique elle peut conduire très vite à des automates de grande taille. Plus d'expérimentation sont nécessaire afin de déterminer l'efficacité du calcul de $A^{(h)}$, notamment dans le cadre de l'analyse en arrière de systèmes réels.

# 8

# Vérification de propriétés LTL sur des graphes de réécriture

**Sommaire**



Cette section présente une utilisation des automates d'arbres à contraintes (section 2.14) pour l'analyse d'accessibilité par réécriture (section 2.4). Nous allons utiliser ces automates pour construire des procédures de semi-decision, basées sur des sur-approximations, pour la vérification de trois modèles de propriétés LTL :

- $\Box(R_1 \Rightarrow \circ R_2)$,
- $\neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$,
- $\Box(R_1 \Rightarrow \Box\neg R_2)$.

Plus précisément, à partir d'un graphe potentiellement infini $M$ des réécritures possibles depuis un langage d'arbres initial et d'une propriété LTL $\varphi$ (suivant un des formats ci-dessus), nous construisons des équations sur les langages permettant de (semi-)vérifier que $M \models \varphi$ (c'est-à-dire que $M$ vérifie la propriété $\varphi$). Étant donné que ce problème de vérification est indécidable, nous définissons des procédures de semi-décision utilisant les sur-approximations et les automates d'arbres à contraintes afin de semi-décider les équations sur les langages précédemment définies.

Dans la section 8.1.2 nous présentons le modèle $R$-LTL permettant de définir des graphes des réécritures possibles. Dans la section 8.2 nous proposons des équations sur les langages permettant le model-checking de trois propriétés LTL. La section 8.3 donne des procédures de semi-décision, pour la vérification de ces trois propriétés, basées sur les sur-approximations et les automates d'arbres à contraintes.

## 8.1 Système de réécriture et LTL

### 8.1.1 État de l'art : propriétés temporelles et réécriture

Il existe actuellement beaucoup de travaux où la LTL [Pnu77] est utilisée pour modéliser et vérifier des propriétés de systèmes. Pour les lecteurs intéressés par ce sujet, nous vous renvoyons à la page web du model-checker Spin [1].

La logique de réécriture [Mes92] est un modèle théorique général permettant de modéliser des systèmes. Dans ce contexte, on considère les graphes de réécriture définis de la manière suivante :

- les nœuds de ces graphes sont étiquetés par des classes d'équivalence d'une théorie équationnelle.
- il y a un lien entre un premier nœud et un deuxième nœud si un élément du premier nœud peut être réécrit en un élément du deuxième nœud.

Si la théorie équationnelle est l'identité, le graphe de réécriture équivaut exactement au système de transition étiqueté défini dans la section 8.1.2. Dans ce domaine, les travaux de [EM07, Mes08, Mes07] s'intéressent aux approches utilisant la LTL. Dans [ACC07], les auteurs proposent un modèle général pour la sécurité des protocoles basé sur les ensembles de réécriture dans un contexte décidable (les graphes sous-jacents sont finis).

### 8.1.2 Le modèle $R$-LTL

Dans cette section, les propriétés temporelles sont définies pour être utilisées dans le contexte de la réécriture. Cette approche est basée sur la LTL [Pnu77]. Le but est d'exprimer et de vérifier des contraintes temporelles sur l'ordre d'application des règles de réécriture pour la relation $\rightarrow_R^*$.

Soit $R$ un système de réécriture et $L_0$ un ensemble de termes. On notera par $G(L_0, R)$ le graphe $R$-étiqueté $(\mathcal{T}(\mathcal{F}), L_0, \Delta)$ où $\Delta = \{t_i \stackrel{l \rightarrow r}{\rightarrow} t_j \mid l \rightarrow r \in R \text{ et } t_j \in \{l \rightarrow r\}(t_i)\}$. Un chemin $\pi$ dans $G(L_0, R)$ est une séquence (finie ou infinie) $(p_1, a_1, q_1) \dots (p_i, a_i, q_i) \dots$ d'éléments de $\Delta$ telle que $p_1 \in L_0$, pour chaque $i \geq 1$ si $p_{i+1}$ existe, alors $q_i = p_{i+1}$. Le mot (fini ou infini) $a_1 \dots a_i \dots$ sur l'alphabet $R$ est appelé label de $\pi$. Un chemin $\pi$ est complet si il est infini ou si il existe un entier $i$ tel que $\pi = (p_1, a_1, q_1), \dots, (p_i, a_i, q_i)$ et $\{p \mid \exists a \in R,\ (q_i, a, p) \in \Delta\}$ est vide.

Une formule LTL $\varphi$ sur $R$ est définie inductivement de la manière suivante :

$$\varphi := \top \mid R_0 \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \circ\varphi_1 \mid \varphi_1 \mathcal{U} \varphi_2$$

où $R_0 \subseteq R$, et $\varphi_1$, $\varphi_2$ sont des formules LTL sur $R$.

De plus on a les définitions d'équivalences de formules LTL suivantes :

$$\Box\varphi = \neg(\top \mathcal{U} \neg\varphi)$$

$$\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$$

$$\varphi_1 \Rightarrow \varphi_2 = (\neg\varphi_1 \vee \varphi_2)$$

Soit $w$ mot fini ou infini sur $R$ (considéré comme un alphabet). La $i^{ième}$ lettre de $w$, si elle existe, est notée $w(i)$. On définit inductivement la satisfaction d'une formule LTL $\varphi$ par $w$ à une position $i$, notée $(w, i) \models \varphi$, par :

1. http://spinroot.com/spin/whatispin.html

| | |
|---|---|
| $(w,i) \models \top$ | si et seulement si $w(i)$ existe, |
| $(w,i) \models R_0$, avec $R_0 \subseteq R$ | si et seulement si $w(i)$ existe et $w(i) \in R_0$, |
| $(w,i) \models \neg\varphi$ | si et seulement si $(w,i) \not\models \varphi$, |
| $(w,i) \models (\varphi_1 \vee \varphi_2)$ | si et seulement si $(w,i) \models \varphi_1$ ou $(w,i) \models \varphi_2$ , |
| $(w,i) \models \circ\varphi$ | si et seulement si $(w,i+1) \models \varphi$, |
| $(w,i) \models (\varphi_1 \mathcal{U} \varphi_2)$ | si et seulement si il existe $j \geq i$ tel que $(w,i) \models \varphi_2$ et pour chaque $i \leq k < j$, $(w,k) \models \varphi_1$. |

On dit que $w$ est un modèle de $\varphi$ si $(w,1) \models \varphi$.

Un graphe $G(L_0,R)$ satisfait une formule $\varphi$, noté $G \models \varphi$, si et seulement si l'étiquette de chaque chemin complet de $G(L_0,R)$ satisfait $\varphi$. Des exemples illustrés sont donnés en section 8.2.

## 8.2 Vérification de trois modèles de propriétés LTL

Dans cette section, nous allons étudier trois motifs formules LTL utiles pour l'expression de propriétés de sécurité pour l'analyse statique d'applications Java.

- La formule $\Box(R_1 \Rightarrow \circ R_2)$ signifie, intuitivement, que si un terme accessible est réécrit en utilisant un règle de $R_1$, alors le terme obtenu peut être réécrit en utilisant uniquement une règle de $R_2$, comme illustré par le graphe abstrait de la figure 8.1.
  Dans le domaine d'application de la programmation, ce motif de formule est utilisé pour exprimer que si une méthode $m_1$ est invoquée, alors une méthode $m_2$ doit être invoquée immédiatement après.
  Par exemple, si la méthode demande à l'utilisateur de s'identifier en utilisant son code PIN, alors la prochaine méthode invoquée doit être l'authentification ou l'annulation de l'authentification.
- La formule $\neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$ est la formule duale de la précédente : si un terme accessible est réécrit en utilisant une règle de $R_2$, alors, à la réécriture précédente, le terme a été réécrit en utilisant une règle de $R_1$, comme illustré par le graphe de la figure 8.2.
  Par exemple, ce motif de formule temporelle peut exprimer le fait que si un SMS est envoyé alors l'utilisateur a précédemment donné son accord.
- La formule $\Box(R_1 \Rightarrow \Box\neg R_2)$ exprime le fait qu'une règle de $R_1$ est utilisée pour réécrire un terme accessible, alors aucune règle de $R_2$ ne peut être utilisé dans le futur, comme le montre la figure 8.3.
  A l'aide de ce motif de formule temporelle, on peut exprimer le fait que si une application particulière accède à des données privées de l'utilisateur (comme son carnet d'adresse par exemple), aucun message ne peut être envoyé par la suite. Ainsi, les données privées de l'utilisateur ne peuvent pas être utilisée à son insu. Notons aussi que, d'après [DAC98], ce motif de formule est utilisé couramment pour la spécification de systèmes.

### 8.2.1 Formule $\Box(R_1 \Rightarrow \circ R_2)$

Dans cette section, nous allons voir comment le model-checking de la formule $\Box(R_1 \Rightarrow \circ R_2)$ peut être traduit en une équation sur les langages. Un graphe $R$-étiqueté satisfaisant cette formule est schématisé par la figure 8.1.

♦ **Proposition 8.1**
*Soit $R$ un système de réécriture, $R_1, R_2 \subseteq R$ et $L_0$ un langage régulier.*
*On a $G(L_0,R) \models \Box(R_1 \Rightarrow \circ R_2)$ si et seulement si*

$$(R \setminus R_2)(R_1(R^*(L_0))) = \emptyset \text{ et } R_1(R^*(L_0)) \cap R_2^{-1}(\mathcal{T}(\mathcal{F})) = R_1(R^*(L_0)).$$

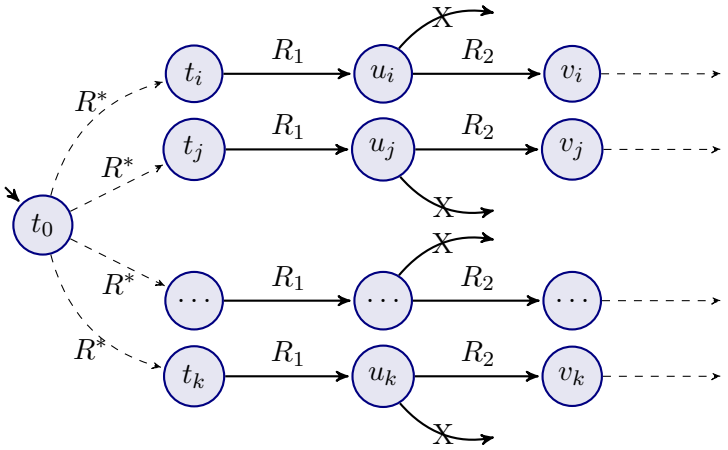



FIGURE 8.1 – Un graphe satisfaisant $\Box(R_1 \Rightarrow \circ R_2)$

*Démonstration.* On suppose que $G(L_0, R) \models \Box(R_1 \Rightarrow \circ R_2)$.
Soit $t$ un terme de $(R \setminus R_2)(R_1(R^*(L_0)))$. Il existe des termes $t_1$ et $t_2$ tels que $t_1 \in R^*(L_0)$ et $t_1 \to_{R_1} t_2 \to_{(R \setminus R_2)} t$.

Comme $t_1 \in R^*(L_0)$, il existe des termes $s_0, \ldots, s_k$ tels que $s_0 \in L_0$, $s_k = t_1$ et $s_{i+1} \in R(s_i)$ pour tout $i < k$. Par conséquent il existe un chemin $(s_0, a_0, s_1) \ldots (s_{k-1}, a_k, t_1)(t_1, a, t_2), (t_2, b, t)$ dans $G(L_0, R)$ tel que $s_0 \in L_0$, $a \in R_1$ et $b \in R \setminus R_2$. Ce chemin peut être étendu à un chemin complet dont l'étiquette ne vérifie pas $\Box(R_1 \Rightarrow \circ R_2)$, ce qui est une contradiction.

Maintenant, comme $R_1(R^*(L_0)) \cap R_2^{-1}(\mathcal{T}(\mathcal{F})) \subseteq R_1(R^*(L_0))$, si $R_1(R^*(L_0)) \cap R_2^{-1}(R^*(L_0)) \neq R_1(R^*(L_0))$, alors il existe $t \in R_1(R^*(L_0))$ tel que $t \notin R_2^{-1}(\mathcal{T}(\mathcal{F}))$.

Il s'en suit qu'il existe un terme $t_1 \in R^*(L_0)$ tel que $t \in R_1(t_1)$. Donc il existe des termes $s_0, \ldots, s_k$ tels que $s_0 \in L_0$, $s_k = t_1$ et $s_{i+1} \in R(s_i)$ pour chaque $i < k$. Par conséquent, il y a un chemin $\pi = (s_0, a_0, s_1) \ldots (s_{k-1}, a_k, t_1)(t_1, b, t)$ dans $G(L_0, R))$ tel que $s_0 \in L_0$, avec $b \in R_1$. Comme $t \notin R_2^{-1}(\mathcal{T}(\mathcal{F}))$, il n'y a pas de terme $t_2$ tel que $t_2 \in R_2(t)$. Par conséquent, $\pi$ ne peut pas être étendu par une transition étiquetée par une partie de $R_2$. Il s'en suit que, soit $\pi$ est maximal et son label ne vérifie pas $\Box(R_1 \Rightarrow \circ R_2)$, ou $\pi$ peut être étendu à un chemin complet qui ne vérifie pas $\Box(R_1 \Rightarrow \circ R_2)$, ce qui est une contradiction.

Et inversement, on suppose que $(R \setminus R_2)(R_1(R^*(L_0))) = \emptyset$ et $R_1(R^*(L_0)) \cap R_2^{-1}(\mathcal{T}(\mathcal{F})) = R_1(R^*(L_0))$.

Soit $\pi = (t_0, a_0, t_1) \ldots (t_k, a_k, t_{k+1}) \ldots$ un chemin maximal dans $G(L_0, R)$ dont l'étiquette ne vérifie pas $\Box(R_1 \Rightarrow \circ R_2)$. On en déduit qu'il existe $i$ tel que $a_i \in R_1$ et soit $a_{i+1} \notin R_2$ ou $a_{i+1}$ n'existe pas (la trace est finie). Si $a_{i+1} \notin R_2$ alors $t_{i+1} \in (R \setminus R_2)(R_1(R^*(L_0)))$, ce qui est une contradiction. Si $a_{i+1}$ n'existe pas, alors $t_{i+1} \in R_1(R^*(L_0))$ mais, d'après le fait que $\pi$ est maximal, $R_2(t_{i+1}) = \emptyset$, ce qui prouve que $t_{i+1} \notin R_2^{-1}(\mathcal{T}(\mathcal{F}))$, ce qui est une contradiction. □

♢ **Exemple 8.2**
*Soit* $\mathcal{F} = \{\bot^0, a^1, b^1, c^1, f^2, g^2\}$ *un ensemble de symboles. Considérons le système de réécriture* $R = \{r_1, r_2, r_3, r_4, r_5\}$ *avec :*

$r_1 = f(b(x), b(x)) \to g(x, x)$,

$r_2 = a(x) \to a(a(x))$,

$r_3 = a(\bot) \to b(\bot)$,

$r_4 = a(b(x)) \to b(b(x))$,
$r_5 = g(x,y) \to c(g(x,y))$.
*Enfin, soit le langage* $L_0 = \{f(a(u(\bot)), v(a(\bot))) \mid u \in \{a,b\}^* \text{ et } v \in a^*\}$.

*On a* $\{r_1\}(R^*(L_0)) \subseteq g(b^*(\bot), b^*(\bot))$. *Ainsi* $(R \backslash \{r_5\})(\{r_1\}(R^*(L_0))) = \emptyset$. *De plus,* $\{r_5\}^{-1}(\mathcal{T}(\mathcal{F}))$ *est l'ensemble des termes où* $g$ *apparaît une fois au moins. Par conséquent,* $\{r_1\}(R^*(L_0)) \cap \{r_5\}^{-1}(\mathcal{T}(\mathcal{F})) = \{r_1\}(R^*(L_0))$. *Et on peut conclure que* $G(L_0, R) \models \Box(\{r_1\} \Rightarrow \circ\{r_5\})$.

### 8.2.2 Formule $\neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$

Dans cette section la formule $\neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$ est retranscrite en équation sur les langages. La figure 8.2 montre un graphe $R$-étiqueté satisfaisant cette formule.

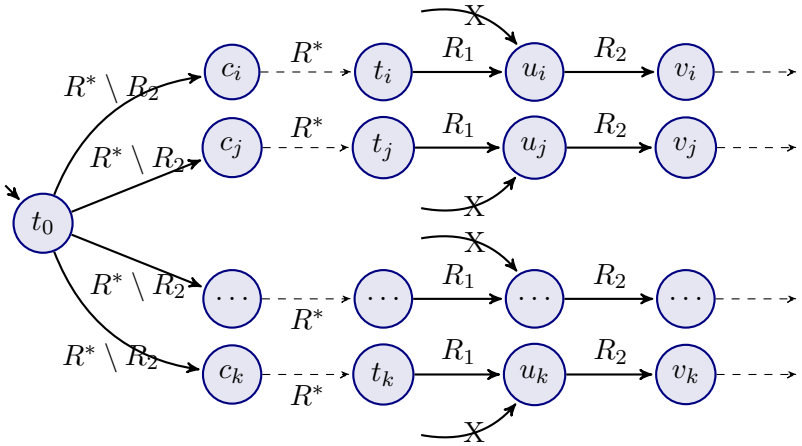



FIGURE 8.2 – Un graphe satisfaisant $\neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$

**♦ Proposition 8.3**
*Soit* $R$ *un système de réécriture,* $R_1, R_2 \subseteq R$ *et* $L_0$ *un langage régulier.*
*On a* $G(L_0, R) \models \neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$ *si et seulement si*

$$R_2((R \setminus R_1)(R^*(L_0))) = \emptyset \text{ et } R_2(L_0) = \emptyset.$$

*Démonstration.* Il est clair que $G(L_0, R) \models \neg R_2$ si et seulement si $R_2(L_0) = \emptyset$. Maintenant nous allons prouver que $G(L_0, R) \models \Box(\circ R_2 \Rightarrow R_1)$ si et seulement si $R_2((R \setminus R_1)(R^*(L_0))) = \emptyset$.

Tout d'abord on suppose que $G(L_0, R) \models \Box(\circ R_2 \Rightarrow R_1)$. Soit $t \in R_2((R \setminus R_1)(R^*(L_0)))$. Il existe deux termes $t_1$ et $t_2$ tels que $t_1 \in R^*(L_0)$ et une règle $a \in R \setminus R_1$ telle que $t_2 \in \{a\}(t_1)$ et $t \in R_2(t_2)$. Comme $t_1 \in R^*(L_0)$, alors il existe des termes $s_0, \ldots, s_k$ tels que $s_0 \in L_0$, $s_k = t_1$ et $s_{i+1} \in R(s_i)$ pour chaque $i < k$. Donc il existe un chemin

$$(s_0, a_0, s_1) \ldots (s_{k-1}, a_k, t_1)(t_1, a, t_2), (t_2, b, t)$$

de $G(L_0, R)$ tel que $s_0 \in L_0$, $a \in R \setminus R_1$, et $b \in R_2$. Comme $a \in R \setminus R_1$, ce chemin peut être étendu à un chemin complet dont l'étiquette ne vérifie pas $\Box(\circ R_2 \Rightarrow R_1)$, ce qui est une contradiction.

Inversement, on suppose que $R_2((R \setminus R_1)(R^*(L_0))) = \emptyset$. Soit

$$\pi = (t_0, a_0, t_1) \ldots (t_k, a_k, t_{k+1}) \ldots$$

un chemin maximal de $G(L_0, R)$ dont l'étiquette ne vérifie pas $\Box(\circ R_2 \Rightarrow R_1)$. Il s'en suit qu'il existe $i$ tel que $a_i \notin R_1$ et $a_{i+1} \in R_2$. Donc on a $t_{i+1} \in R_2((R \setminus R_1)(R^*(L_0))) \neq \emptyset$, ce qui est une contradiction. □

♢ **Exemple 8.4**
*Avec les mêmes données que pour l'exemple 8.2, on a* $\{r_5\}(L_0) = \emptyset$. *De plus, on peut vérifier que g n'apparaît pas dans les termes* $R \setminus \{r_1, r_5\}(R^*(L_0))$, *prouvant que* $\{r_5\}(R \setminus \{r_1, r_5\}(R^*(L_0))) = \emptyset$. *Par conséquent,* $G(L_0, R) \models \neg\{r_5\} \wedge \Box(\circ\{r_5\} \Rightarrow \{r_1, r_5\})$.

### 8.2.3 Formule $\Box(R_1 \Rightarrow \Box\neg R_2)$

Dans cette section nous allons voir comment vérifier la satisfiabilité de la formule $\Box(R_1 \Rightarrow \Box\neg R_2)$ à l'aide d'équations sur les langages. La figure 8.3 montre un graphe $R$-étiqueté satisfaisant cette formule.

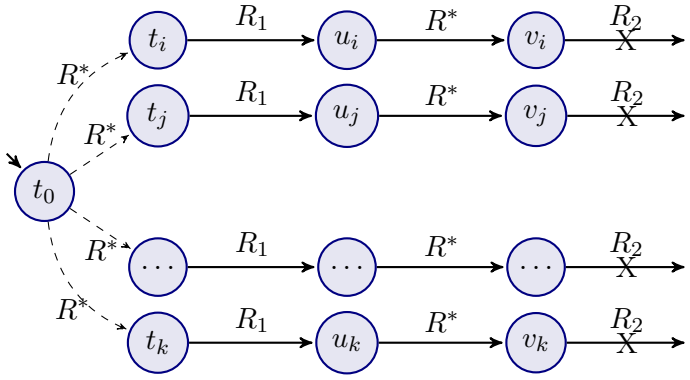



FIGURE 8.3 – Un graphe satisfaisant $\Box(R_1 \Rightarrow \Box\neg R_2)$

♦ **Proposition 8.5**
*Soit $R$ un système de réécriture, $R_1, R_2 \subseteq R$ et $L_0$ un langage régulier.*
*On a $G(L_0, R) \models \Box(R_1 \Rightarrow \Box\neg R_2)$ si et seulement si*

$$R_2(R^*(R_1(R^*(L_0)))) = \emptyset.$$

*Démonstration.* Tout d'abord on suppose que $R_2(R^*(R_1(R^*(L_0)))) \neq \emptyset$. Soit $t$ un terme de $R_2(R^*(R_1(R^*(L_0))))$. Il existe des termes $t_0$, $t_1$, $t_2$ tels que $t_0 \in R^*(L_0)$, et $t_0 \to_{R_1} t_1 \to_R^* t_2 \to_{R_2} t$. Ce qui implique qu'il existe un chemin complet $w$ dans $G(L_0, R)$ tel que $(w, 1) \not\models \Box(R_1 \Rightarrow \Box\neg R_2)$.

Maintenant on suppose que $R_2(R^*(R_1(R^*(L_0)))) = \emptyset$. Soit $\pi = (t_0, a_0, t_1) \dots (t_k, a_k, t_{k+1}) \dots$ un chemin maximal de $G(L_0, R)$ dont l'étiquette ne vérifie pas $\Box(R_1 \Rightarrow \Box\neg R_2)$. Il s'en suit qu'il existe $i$ tel que $a_i \in R_1$ et $j > i$ tel que $a_j \in R_2$. Donc $t_{j+1} \in R_2(R^*(R_1(R^*(L_0))))$, ce qui est une contradiction. □

♢ **Exemple 8.6**
*En reprenant les données de l'exemple 8.2, on a* $\{r_1\}(R^*(L_0)) \subseteq g(b^*(\bot), b^*(\bot))$. *Il apparaît que a n'est dans aucun terme de* $R^*(\{r_1\}(R^*(L_0)))$. *Par conséquent,* $\{r_2\}(R^*(\{r_1\}(R^*(L_0)))) = \emptyset$, *prouvant que* $G(L_0, R) \models \Box(\{r_1\} \Rightarrow \Box\neg\{r_2\})$.

## 8.3 Procédures de semi-décisions

Dans la section 8.3.1, nous allons tout d'abord montrer que pour les propriétés précédentes le model-checking est indécidable. Pour obtenir des procédures de semi-décision pour le model-cheking de ces propriétés, nous allons présenter une construction basée sur les automates à contraintes. Dans la section 8.3.2, nous allons voir comment exploiter les sur-approximations et la construction expliquée en section 8.3.1 pour vérifier les trois propriétés présentées dans la section 8.2.

**Notes sur les notations.** Dans cette section la notation pour les automates d'arbres et pour les TAGED sera différentes que précédemment pour des raisons de clarté.
Un automate d'arbres $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta)$ sera noté $A = (\mathcal{Q}, F, \Delta)$ où $F = \mathcal{Q}_f$.
Un TAGED $A = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta, E)$ sera noté $A = (\mathcal{Q}, E, F, \Delta)$.

### 8.3.1 Réécriture et TAGED

Nous allons tout d'abord montrer que le model-checking des trois motifs de formules temporelles est indécidable.

♦ **Proposition 8.7**
*Soit un système de réécriture $R$, $R_1, R_2 \subseteq R$ et un terme $t_0$, on ne peut pas décider si $G(\{t_0\}, R)) \models \Box(R_1 \Rightarrow \circ R_2)$ (respectivement $G(\{t_0\}, R) \models \Box(\circ R_2 \Rightarrow R_1)$) (respectivement $G(\{t_0\}, R)) \models \Box(R_1 \Rightarrow \Box\neg R_2)$).*

*Démonstration.* Il est bien connu que le problème suivant, appelé **Atteignabilité**$(R, s, t, \mathcal{F})$ est indécidable.
**Données initiales :** Un système de réécriture $R$ sur $\mathcal{T}(\mathcal{F})$, deux termes $s$ et $t$ de $\mathcal{T}(\mathcal{F})$.
**Problème :** Est-ce que $s \rightarrow_R^* t$ ?

On suppose qu'il existe un algorithme $P_1(R, R_1, R_2, L_0, \mathcal{F})$ qui, d'après un système de réécriture $R$ et un ensemble de termes $L_0$ de $\mathcal{T}(\mathcal{F})$, décide de $G(L_0, R) \models \Box(R_1 \Rightarrow \circ R_2)$. Soit $R_0, s_0, t_0, \mathcal{F}_0$ une instance du problème d'atteignabilité. Soit $\#, \$ \notin \mathcal{F}_0$ et $\mathcal{F}_1 = \mathcal{F} \cup \{\#, \$\}$, avec $ar(\#) = ar(\$) = 0$. On dit que $P_1(R_0 \cup \{t_0 \rightarrow \#, \$ \rightarrow \#\}, \{t_0 \rightarrow \#\}, \{\$ \rightarrow \#\}, \{s_0\}, \mathcal{F}_1) = \mathsf{false}$ si et seulement si **Atteignabilité**$(R_0, s_0, t_0, \mathcal{F}_0)$=true.

En effet, si **Atteignabilité**$(R_0, s_0, t_0, \mathcal{F}_0)$=true, alors il existe dans $G(\{s_0\}, R_0)$ un chemin $\pi$ de $s_0$ à $t_0$. Par construction, $\pi$ est aussi un chemin de $G(\{s_0\}, R \cup \{t_0 \rightarrow \#, \$ \rightarrow \#\})$. Mais $\pi$,$(t_0, \{t_0 \rightarrow \#\}, \#)$ chemin complet de $G(\{s_0\}, R \cup \{t_0 \rightarrow \#, \$ \rightarrow \#\})$ dont l'étiquette ne satisfait pas $\{t_0 \rightarrow \#\} \Rightarrow \circ\{\$ \rightarrow \#\}$. Par conséquent $P_1(R_0 \cup \{t_0 \rightarrow \#, \$ \rightarrow \#\}, \{t_0 \rightarrow \#\}, \{\$ \rightarrow \#\}, \{s_0\}, \mathcal{F}_1) = \mathsf{false}$.

Et inversement, si $P_1(R_0 \cup \{t_0 \rightarrow \#, \$ \rightarrow \#\}, \{t_0 \rightarrow \#\}, \{\$ \rightarrow \#\}, \{s_0\}, \mathcal{F}_1) = \mathsf{false}$, alors il existe un chemin complet $\pi'$ dans $G(\{s_0\}, R \cup \{t_0 \rightarrow \#, \$ \rightarrow \#\})$ dont l'étiquette ne satisfait pas $\{t_0 \rightarrow \#\} \Rightarrow \circ\{\$ \rightarrow \#\}$. Donc, la transition $\{t_0 \rightarrow \#\}$ est utilisée dans $\pi'$. Il s'en suit que $t_0$ est atteignable dans $G(\{s_0\}, R \cup \{t_0 \rightarrow \#, \$ \rightarrow \#\})$ depuis $s_0$. Il est clair que **Atteignabilité**$(R_0, s_0, t_0, \mathcal{F}_0)$=true, ce qui conclue la preuve.

La preuve d'indécidabilité pour les deux autres formules peut être faite à l'aide de réductions similaires. □

Maintenant nous allons fournir des outils théoriques, basés sur les automates à contraintes, afin d'aider à la construction de semi-algorithmes en rapport avec les égalités sur les langages de la section 8.2.

♦ **Proposition 8.8**
*Soit $R$ un système de réécriture. On peut calculer en temps polynomial un TAGED positif acceptant la langage $R^{-1}(\mathcal{T}(\mathcal{F}))$.*

*Démonstration.* Soit $l \to r \in R$. Soit $A_l = (\mathcal{Q}_l, E_l, F_l, \Delta_l)$ un TAGED positif défini par :
- $\mathcal{Q}_l = \{q_i \mid i \in \mathcal{P}os_{\mathcal{F}}(l)\} \cup \{q_x, q_x^a \mid x \in \mathcal{V}ar(l)\} \cup \{q^a\}$,
- $E_l = \{(q_x, q_x) \mid x \in \mathcal{V}ar(l)\}$,
- $\Delta_l = \Delta_1 \cup \Delta_2$ avec $\Delta_1 = \{l(p)(q_{\alpha_1}, \ldots, q_{\alpha_n}) \to q_p \mid p \in \mathcal{P}os(l)$ et $\alpha_i = p.i$ si $l(p.i) \in \mathcal{F}$ et $\alpha_i = x$ sinon$\} \cup \{f(q_x^a, \ldots, q_x^a) \to q_x^a \mid f \in \mathcal{F},\ x \in \mathcal{V}ar(l)\} \cup \{f(q_x^a, \ldots, q_x^a) \to q_x \mid f \in \mathcal{F},\ x \in \mathcal{V}ar(l)\}$ et
  $\Delta_2 = \{f(q^a, \ldots, q^a) \to q^a \mid f \in \mathcal{F}\} \cup \{f(q^a, \ldots, q^a, q_\varepsilon, q^a, \ldots, q^a) \to q_\varepsilon \mid f \in \mathcal{F}\}$,
- $F_l = \{q_\varepsilon\}$.

Notons tout d'abord que $\{t \mid t \to_{A_l}^* q_x\} = \mathcal{T}(\mathcal{F})$. Ensuite, $\{t \mid t \to_{\Delta_l}^* q_\varepsilon\} = \{t \mid \exists p \in \mathcal{P}os(t), \mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})\ s.t.\ t|_p = l\mu\}$. Il s'en suit que $L(A_l) = \{l \to r\}^{-1}(\mathcal{T}(\mathcal{F}))$. On voit que cette construction s'effectue en un temps polynomial ($\mathcal{F}$ est considéré comme fixe et n'est pas un paramètre du problème). La complexité polynomiale découle directement de [FTT08, Proposition 2]. Cependant la complexité est exponentielle en fonction de l'arité maximale des symboles de $\mathcal{F}$. □

Notons que si $R$ est linéaire à gauche, alors le TAGED positif obtenu est un automate d'arbres où, pour toute variable $x$, l'état $q_x$ apparaît une fois au plus dans une exécution.

Voici un exemple de la construction décrite dans la preuve précédente.

♢ **Exemple 8.9**
*Soit $\mathcal{F} = \{\bot^0, h^1, f^2\}$ un ensemble de symboles. Le langage $\{f(x,x) \to h(x)\}^{-1}(\mathcal{T}(\mathcal{F}))$ est reconnu par le TAGED positif $A_l = (\mathcal{Q}_l, E_l, F_l, \Delta_l)$ avec*
- $\mathcal{Q}_l = \{q_\varepsilon, q_1, q_2\} \cup \{q_x, q_x^a\} \cup \{q^a\}$,
- $E_l = \{(q_x, q_x)\}$,
- $F_l = \{q_\varepsilon\}$,
- $\Delta_l = \Delta_1 \cup \Delta_2$ *avec* $\Delta_1 = \{f(q_x, q_x) \to q_\varepsilon\} \cup \{f(q_x^a, q_x^a) \to q_x^a, \bot \to q_x^a, h(q_x^a) \to q_x^a\} \cup \{f(q_x^a, q_x^a) \to q_x, \bot \to q_x, h(q_x^a) \to q_x\}$ *et* $\Delta_2 = \{f(q^a, q^a) \to q^a, \bot \to q_a, h(q_a) \to q_a\} \cup \{f(q^a, q_\varepsilon) \to q_\varepsilon, f(q_\varepsilon, q^a) \to q_\varepsilon, h(q_\varepsilon) \to q_\varepsilon\}$.

♦ **Proposition 8.10**
*Soit $A$ un TAGED positif et $R$ un système de réécriture. Décider que $R(L(A))$ est vide est de complexité EXPTIME.*

*Démonstration.* On remarque que $R(L(A))$ est vide si et seulement si $L(A) \cap R^{-1}(\mathcal{T}(\mathcal{F})) = \emptyset$. La proposition est alors une conséquence directe de la proposition 8.8 et du fait que les langages reconnaissables par des TAGED sont clos pour l'intersection [FTT08, Proposition 2] et que le problème de vacuité pour les TAGED positifs se résout en temps exponentiel [FTT08, Theorem 1]. □

♦ **Proposition 8.11**
*Soit $A$ un automate d'arbres et $R$ un système de réécriture. Le langage $R(L(A))$ est accepté par un TAGED positif.*

*Démonstration.* On note que la preuve est constructive et l'exemple 8.12 illustre cette propriété.

Étant donné que $R(L(A)) = \cup_{l \to r \in R}\{l \to r\}(L(A))$ et comme les langages reconnus par des TAGED positifs sont clos pour l'union, il suffit de prouver la proposition pour une seule règle de réécriture $l \to r$.

La preuve est décomposée en trois parties :

**1re partie** on propose tout d'abord la construction de TAGED positifs $A_{r,\sigma,q}$ utiles à la démonstration.

Ensuite on prouve que $\{l \to r\}(L(A))$ est accepté par une union (finie) des $L(A_{r,\sigma,q})$ en montrant

**2e partie** que $L(A_{r,\sigma,q}) \supseteq \{l \to r\}(L(A))$,

**3e partie** et que $L(A_{r,\sigma,q}) \subseteq \{l \to r\}L(A)$.

Comme la classe de langages acceptés par les TAGED positifs sont clos pour l'union, la preuve est complète une fois ces trois points montrés.

**1re partie.** Soit $l \to r \in R$. On définit une $(l \to r)$-substitution comme une application de $\mathcal{P}os_{\mathcal{X}}(l)$ dans $\mathcal{Q}$. Soit $\sigma$ une $(l \to r)$-substitution. On note par $l\sigma$ le terme de $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ défini par : $\mathcal{P}os(l\sigma) = \mathcal{P}os(l)$, et pour chaque $p \in \mathcal{P}os(l)$, si $p \in \mathcal{P}os_{\mathcal{X}}(l)$ alors $l\sigma(p) = \sigma(l(p))$, ou sinon $l\sigma(p) = l(p)$.

Soit l'automate $A = (\mathcal{Q}, F, \Delta)$. Comme la classe des langages d'arbres réguliers est clos pour l'intersection, pour chaque variable $x$ apparaissant dans $l$ et pour chaque $(l \to r)$-substitution $\sigma$, il existe un automate d'arbres fini $A^\sigma_x = (\mathcal{Q}^\sigma_x, F^\sigma_x, \Delta^\sigma_x)$ tel que

$$L(A^\sigma_x) = \bigcap_{p \in \mathcal{P}os_{\{x\}}(l)} L(A, \sigma(p)).$$

On suppose, sans perte de généralité, que les états de $F^\sigma_x$ n'apparaissent pas dans la partie gauche des transitions de $\Delta^\sigma_x$.

Soit $A_{r,\sigma,q} = (\mathcal{Q}_{r,\sigma,q}, E_{r,\sigma,q}, F_{r,\sigma,q}, \Delta_{r,\sigma,q})$ un TAGED positif défini par :
- $\mathcal{Q}_{r,\sigma,q} = \mathcal{Q} \cup \{q_i \mid i \in \mathcal{P}os_{\mathcal{F}}(r)\} \cup \{q^+ \mid q \in \mathcal{Q}\} \cup \bigcup_{x \in \mathcal{V}ar(r)} \mathcal{Q}^\sigma_x$,
- $E_{r,\sigma,q} = \{(q^1, q^2) \mid \exists x \in \mathcal{V}ar(r)$ telle que $q^1, q^2 \in F^\sigma_x\}$,
- $F_{r,\sigma,q} = \{q^+_f \mid q_f \in F\}$,
- $\Delta_{r,\sigma,q} = \Delta \cup \Delta_1 \cup \Delta_2$ avec
  $\Delta_1 = \{r(p)(q_{\alpha_1}, \ldots, q_{\alpha_n}) \to q_p \mid p \in \mathcal{P}os(r)$ et $\alpha_i = p.i$ si $r(p.i) \in \mathcal{F}$ et $q_{\alpha_i} \in F^\sigma_{r(p.i)}$ sinon$\} \cup$
  $\bigcup_{x \in \mathcal{V}ar(r)} \Delta^\sigma_x$
  $\Delta_2 = \{f(s_1, \ldots, s_j, q_\varepsilon, s_{j+1}, \ldots, s_n) \to s^+_{n+1} \mid s_i \in \mathcal{Q}$ et
  $f(s_1, \ldots, s_{j-1}, q, s_{j+1}, \ldots, s_n) \to s_{n+1} \in \Delta\}$
  $\cup \{f(s_1, \ldots, s_{j-1}, s^+_j, s_{j+1}, \ldots, s_n) \to s^+_{n+1} \mid s_i \in \mathcal{Q}$ et
  $f(s_1, \ldots, s_{j-1}, s_j, s_{j+1}, \ldots, s_n) \to s_{n+1} \in \Delta,\ Arity(f) \geq 1\}$.

On déclare que

$$R(L(A)) = \bigcup_{l\sigma \to^*_A q} L(A_{r,\sigma,q}),$$

où l'union est réalisée pour chaque état $q \in Q$, chaque $(l \to r)$-substitution $\sigma$ tel que $l\sigma \to^*_A q$ et $L(A^\sigma_x) \neq \emptyset$ pour tout $x \in \mathcal{V}ar(l)$.

**2e partie** On suppose que $t \in R(L(A))$. Il existe un terme $t_0 \in L(A)$, une substitution $\mu$ de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F})$ et une position $p$ de $t_0$ tels que

$$t_0 = t_0[l\mu]_p \quad \text{et} \quad t = t_0[r\mu]_p. \tag{8.1}$$

Soit $\{p_1, \ldots, p_k\} = \mathcal{P}os_{\mathcal{X}}(l)$. Comme $t_0 \in L(A)$ il existe $q, q_1, \ldots, q_k \in \mathcal{Q}$ tels que

$$l\mu \to^*_A l[q_1]_{p_1} \ldots [q_k]_{p_k} \to^*_A q \quad \text{et} \quad t_0[q]_p \to^*_A q_f \in F. \tag{8.2}$$

Soit $\sigma$ une $(l \to r)$-substitution définie par $\sigma(p_i) = q_i$. Par construction on a pour chaque $x \in \mathcal{V}ar(l)$,

$$\mu(x) \in \bigcap_{p \in \mathcal{P}os_{\{x\}}(l)} L(A, \sigma(p)). \tag{8.3}$$

Par définition de $A^\sigma_x$ on a ensuite

$$\mu(x) \in L(A^\sigma_x). \tag{8.4}$$

Il s'en suit que pour chaque $x \in \mathcal{V}ar(r)$,

$$\mu(x) \to^*_{\Delta_1} q_x \in F^\sigma_x \tag{8.5}$$

Par conséquent,

$$r\mu \to^*_{\Delta_1} q_\varepsilon \tag{8.6}$$

En utilisant (8.1) et (8.2) on peut déduire que

$$t \to^*_{\Delta_1} t_0[q]_p \to^*_{\Delta_2} q^+_f, \tag{8.7}$$

ce qui prouve que $t \in L(A_{r,\sigma,q})$. On note que la contrainte définie par $E_{r,\sigma,q}$ est satisfaite si, durant la réduction de $t \to^*_{\Delta_1} t_0[q]_p$, deux états $q^1, q^2 \in F^\sigma_x$ sont utilisés aux positions $p'_1$ et $p'_2$, alors $t_{|p'_1} = t_{|p'_2} = \mu(x)$.

**3e partie** On suppose que $t \in L(A_{r,\sigma,q})$ pour un état $q \in Q$ et une $(l \to r)$-substitution $\sigma$ tels que $l\sigma \to^*_A q$ et $L(A^\sigma_x) \neq \emptyset$ pour chaque $x \in \mathcal{V}ar(l)$. Soit $\rho$ une exécution réussie de $A_{r,\sigma,q}$ sur $t$. On voit qu'il existe une unique position $p$ de $t$ telle que $\rho(p) = q_\varepsilon$. Soit $\{p_1, \ldots, p_k\} = \mathcal{P}os_{\mathcal{X}}(r)$. Par définition de $E_{r,\sigma,q}$, si $\rho(p_i), \rho(p_j) \in F^\sigma_x$ pour une variable $x$ apparaissant dans $r$, alors $r_{|p_i} = r_{|p_j}$. Donc on peut définir une substitution $\mu$, de $\mathcal{V}ar(r)$ dans $\mathcal{T}(\mathcal{F})$, de la manière suivante : si $\rho(p_i) \in F^\sigma_x$, alors $\mu(x) = r_{|p_i}$. Grâce à cette construction on a

$$\mu(x) \in L(A^\sigma_x) \tag{8.8}$$

et

$$t = t[r\mu]_p. \tag{8.9}$$

On rappelle que $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, la substitution $\mu$ est étendue à $\mathcal{V}ar(l)$ de la manière suivante : si $z \in \mathcal{V}ar(l)$ et $z \notin \mathcal{V}ar(r)$, soit $\mu(z)$ un élément arbitrairement choisi dans $L(A^\sigma_x)$ (qui est par hypothèse non vide). Par conséquent, pour chaque $x \in \mathcal{V}ar(l)$ et chaque position $p_x$ de $l$ telle que $l(p_x) = x$,

$$\mu(x) \to^*_A \sigma(p_x) \tag{8.10}$$

Ainsi

$$l\mu(x) \to^*_A l\sigma \tag{8.11}$$

Comme $t \in L(A_{r,\sigma,q})$ on a

$$t \to^*_{\Delta_1} t[q_\varepsilon]_p \to^*_{\Delta_2} q^+_f \quad \text{avec } q^+_f \in F_{r,\sigma,q}. \tag{8.12}$$

Comme $t[q_\varepsilon]_p \to^*_{\Delta_2} q^+_f$,

$$t[q]_p \to^*_A q_f. \tag{8.13}$$

En utilisant (8.11) et (8.13) on a

$$t[l\mu]_p \to^*_A t[l\sigma]_p \to^*_A t[q]_p \to^*_A q_f. \tag{8.14}$$

Par conséquent $t \in R(L(A))$, ce qui prouve la propriété. □

♢ **Exemple 8.12**
*Soit* $\mathcal{F} = \{\bot^0, a^1, b^1, f^2\}$ *un ensemble de symboles. On considère l'automate d'arbres* $A = (\mathcal{Q}, F, \Delta)$ *tel que :*

- $\mathcal{Q} = \{s_0, s_1, s_2, s_4, s_f\}$
- $F = \{s_f\}$
- $\Delta = \{$ $\bot \to s_0$ $\quad$ $\bot \to s_2$ $\quad$ $f(s_1, s_3) \to s_4$
  $a(s_0) \to s_0$ $\quad$ $b(s_2) \to s_3$ $\quad$ $f(s_3, s_4) \to s_f$ $\}$
  $b(s_0) \to s_0$ $\quad$ $a(s_3) \to s_3$
  $a(s_0) \to s_1$ $\quad$ $b(s_3) \to s_3$

*Les termes de l'ensemble* $L_1 = a(\{a, b\}^*(\bot))$ *peuvent être réduit à l'état* $s_1$.
*Les termes de l'ensemble* $L_2 = \{a, b\}^*(b(\bot))$ *peuvent être réduit à l'état* $s_3$.
*Le langage reconnu par l'automate d'arbres* $A$ *est de la forme* $f(L_2, f(L_1, L_2))$.

*Soit* $R = \{f(x, x) \to a(f(x, b(x))\}$ *un système de réécriture. Nous allons construire un TA-GED reconnaissant le langage* $R(L(A))$ *en utilisant la méthode décrite dans la preuve de la proposition 8.11.*

*La seule variable apparaissant dans* $f(x, x)$ *est la variable* $x$. *Alors nous recherchons les substitutions telles que* $L(A, \sigma(1)) \cap L(A, \sigma(2)) \neq \emptyset$ *et que* $f(\sigma(1), \sigma(2)) \to_A^* q$, *où* $q$ *est un état de* $\mathcal{Q}$. *D'après cette dernière condition nous avons deux substitutions* $\sigma_0$ *et* $\sigma_1$ *définies par* $\sigma_0(1) = s_1$, $\sigma_0(2) = s_3$ *et* $\sigma_1(1) = s_3$, $\sigma_1(2) = s_4$.

*Comme la première condition sur l'intersection des langages n'est pas satisfaites pour la substitution* $\sigma_1$, *on obtient :*

$$R(L(A)) = L(A_{a(f(x,b(x))),\sigma_0,s_4}).$$

*De plus, on a* $L(A, \sigma_0(1)) \cap L(A, \sigma_0(2)) = L_1 \cap L_2 = a(\{a, b\}^*(b(\bot)))$. *L'automate d'arbres* $A_x^{\sigma_0} = (\mathcal{Q}_x^{\sigma_0}, F_x^{\sigma_0}, \Delta_x^{\sigma_0})$ *est défini par :*

- $\mathcal{Q}_x^{\sigma_0} = \{s_5, s_6, s_7\}$
- $F_x^{\sigma_0} = \{s_7\}$
- $\Delta_x^{\sigma_0} = \{\bot \to s_5, b(s_5) \to s_6, a(s_6) \to s_6, b(s_6) \to s_6, a(s_6) \to s_7\}$

*L'automate* $A_{a(f(x,b(x))),\sigma_0,s_4} = (\mathcal{Q}_{r,\sigma_0,s_4}, E_{r,\sigma_0,s_4}, F_{r,\sigma_0,s_4}, \Delta_{r,\sigma_0,s_4})$ *(où* $r = a(f(x, b(x)))$*) est défini par :*

- $\mathcal{Q}_{r,\sigma_0,s_4} = \{s_0, s_1, s_2, s_3, s_4, s_f\} \cup \{q_1, q_{1.2}, q_\varepsilon\} \cup \{s_0^+, s_1^+, s_2^+, s_3^+, s_4^+, s_f^+\} \cup \{s_5, s_6, s_7\}$,
- $E_{r,\sigma_0,s_4} = (s_7, s_7)$,
- $F_{r,\sigma_0,s_4} = \{s_f^+\}$,
- $\Delta_{r,\sigma_0,s_4} = \Delta \cup \Delta_1 \cup \Delta_2$, *avec* $\Delta_1 = \{b(s_7) \to q_{1.2}, f(s_7, q_{1.2}) \to q_1, a(q_1) \to q_\varepsilon\} \cup \{\bot \to s_5, b(s_5) \to s_6, a(s_6) \to s_6, b(s_6) \to s_6, a(s_6) \to s_7\}$ *et* $\Delta_2$ *est l'union de* $\{f(s_3, q_\varepsilon) \to s_f^+\}$ *et de l'ensemble de transitions :*
  $a(s_0^+) \to s_0^+$ $\quad$ $b(s_2^+) \to s_3$ $\quad$ $f(s_3^+, s_4) \to s_f^+$
  $b(s_0^+) \to s_0^+$ $\quad$ $a(s_3^+) \to s_3$ $\quad$ $f(s_1, s_3^+) \to s_4^+$
  $a(s_0^+) \to s_1^+$ $\quad$ $b(s_3^+) \to s_3$ $\quad$ $f(s_3, s_4^+) \to s_f^+$
  $f(s_1^+, s_3) \to s_4^+$

### 8.3.2 Semi-algorithmes

Afin de semi-décider qu'une propriété temporelle est satisfaite ou non, nous allons avoir besoin des procédures suivantes :

- $\mathsf{Approx}(A,R)$, où $A$ est un automate d'arbres et $R$ est un système de réécriture, retourne un automate d'arbres $B$ tel que $R^*(L(A)) \subseteq L(B)$. Ceci peut être calculé en utilisant la procédure définie dans [BHK09].
- $\mathsf{ta}(A)$, où $A$ est un TAGED positif, retourne l'automate d'arbres $ta(A)$.
- $\mathsf{OneStep}(A,R)$, où $A$ est un automate d'arbres et $R$ est un système de réécriture, retourne le TAGED positif $B$ acceptant le langage $R(L(A))$ construit à partir de la proposition 8.11.
- $\mathsf{Backward}(R)$, où $R$ est un système de réécriture, retourne un TAGED positif $B$ acceptant $R^{-1}(\mathcal{T}(\mathcal{F}))$ construit à partir de la proposition 8.8.
- $\mathsf{IsEmpty}(A,R)$, où $A$ est un TAGED positif $R$ est un système de réécriture, retourne $\mathsf{vrai}$ si $R(L(A))$ est vide et $\mathsf{faux}$, sinon.

Les procédures ci-dessus et les résultats de la section 8.2 permettent de déduire les résultats suivants.

♦ **Proposition 8.13**
*Soit $R$ un système de réécriture, $R_1, R_2 \subseteq R$ et $A$ un automate d'arbres.*
*Nous avons les propriétés suivantes :*

**(1)** *Si $R_2$ est linéaire à gauche et si* $\mathsf{IsEmpty}(\mathsf{OneStep}(\mathsf{Approx}(A,R),R_1),R \setminus R_2) = \mathsf{vrai}$ *et si* $\mathsf{OneStep}(\mathsf{Approx}(A,R),R_1) \subseteq \mathsf{Backward}(R_2)$,
*alors $G(L(A),R) \models \Box(R_1 \Rightarrow \circ R_2)$.*

**(2)** *Si* $\mathsf{IsEmpty}(A,R_2)$ *et si* $\mathsf{IsEmpty}(\mathsf{OneStep}(\mathsf{Approx}(A,R),R \setminus R_1),R_2) = \mathsf{true}$,
*alors $G(L(A),R) \models \Box(\circ R_2 \Rightarrow R_1)$.*

**(3)** *Si* $\mathsf{IsEmpty}(\mathsf{Approx}(\mathsf{ta}(\mathsf{OneStep}(\mathsf{Approx}(A,R),R_1)),R),R_2) = \mathsf{true}$,
*alors $G(L(A),R) \models \Box(R_1 \Rightarrow \Box\neg R_2)$.*

Notons que dans le point **(1)**, $R_2$ doit être linéaire à gauche pour que le test de l'inclusion soit décidable.

## 8.4 Bilan et perspectives

Dans cette section nous avons proposé d'exploiter la technique de réécritures d'approximations (section 2.4) pour vérifier des propriétés exprimées à l'aide de formules LTL. Cette approche permet de vérifier ses propriétés sur des systèmes infinis et est basée sur la réécriture d'approximation et les TAGED positifs. Sur cette base, nous avons fournis trois procédures de semi-décision afin de vérifier trois modèles de propriétés LTL.

Pour le moment ces procédures n'ont pas été implémenté, on ne peut donc pas conclure sur leur efficacité. La complétion étant déjà implémenté dans les outils Timbuk et Tom, un travail futur pourrait d'intégrer la gestion des TAGED dans ces outils. De plus il pourrait être intéressant d'étendre cette méthode de vérification à d'autres formules LTL, voir la généraliser à l'ensemble (ou un sous-ensemble) des formules LTL.

# Quatrième partie

# Conclusion

Le model-checking régulier permet la vérification de propriétés pour des systèmes comprenant un nombre d'états infini, appelés états atteignables. A l'aide d'un algorithme de complétion d'automate d'arbres il est possible calculer une représentation finie d'une sur-approximation des états atteignables d'un système, ce qui permet ensuite de vérifier la non-atteignabilité d'états indésirables. Cette sur-approximation est calculée à l'aide d'une fonction d'approximation, définie arbitrairement par un utilisateur. Basée sur cette technique, cette thèse avait comme objectif d'étendre le domaine d'application de cette technique et de l'automatiser davantage.

## Le raffinement d'approximation

Nous avons présenté dans cette thèse une technique de raffinement d'approximation permettant :

- la modification automatique de la fonction d'approximation (et donc de la sur-approximation) dans le cas où des états indésirables font parti de celle-ci mais ne font pas parti des états atteignables,
- la vérification de l'atteignabilité d'un état indésirable.

Ainsi la définition manuelle de la fonction d'approximation n'est plus nécessaire car celle-ci peut-être corrigé automatiquement par le raffinement d'approximation. De plus il est maintenant possible de déterminer si un état indésirable est atteignable ou non pour un système donné. Un prototype de l'algorithme de raffinement d'approximation a été intégré à l'outil Timbuk.

L'application du raffinement d'approximation nécessite un système de réécriture linéaire, contrairement à la complétion d'automate d'arbres qui nécessite un système de réécriture linéaire à gauche. Dans des cas pratiques, comme la vérification de protocoles de sécurité ou de bytecode Java, la présence de règle de réécritures non-linéaire à droite empêche donc l'application du raffinement.

C'est dans ce cadre là que nous avons proposé une solution permettant d'appliquer le raffinement d'approximation (et plus généralement la complétion d'automate d'arbres) à des systèmes de réécriture non-linéaires à gauche autrement qu'en déterminisant les automates d'arbres (opération exponentielle). Cette solution consiste à modifier l'automate d'arbres à l'aide d'un algorithme polynomial, en fonction du système de réécriture, afin de pouvoir appliquer la complétion d'automate d'arbres. Cependant le manque d'expérimentation ne permet pas de conclure quand à l'efficacité réelle de cette technique.

## Vérification de propriété LTL

Le problème du model-checking régulier consiste à savoir si un terme $t$ d'un ensemble initial régulier $I$ peut être réécrit en un terme appartenant à un ensemble de terme indésirable $P$ à l'aide d'un ensemble de relations de réécriture représentant les actions possibles du modèle.

Dans cette thèse nous avons proposé une solution permettant de vérifier trois propriétés LTL exprimant des contraintes sur l'ordre d'application des actions :

- $\Box(R_1 \Rightarrow \circ R_2)$,
- $\neg R_2 \wedge \Box(\circ R_2 \Rightarrow R_1)$,
- $\Box(R_1 \Rightarrow \Box\neg R_2)$.

Ces propriétés sont vérifiées sur des graphes de réécriture exprimant toutes les réécritures possibles d'un ensemble de termes par un système de réécriture. Les semi-algorithmes de vérification de ces trois propriétés sont issus d'équations sur les langages semi-décidables à l'aide de la complétion d'automate d'arbres et de TAGED positifs.

Ces semi-algorithmes n'ont pas été expérimentés ni implantés.

## Applications de la réécriture d'approximation

Nous avons pu voir deux applications du model-checking régulier par réécriture de sur-approximation : la vérification de spécifications CCS (sans renommage) et la vérification de deux problèmes pour les machines de Turing.

Les automates d'arbres et les systèmes de réécriture ayant un grand pouvoir d'expression, on peut imaginer qu'il existe d'autres applications intéressantes de cette technique. Le principal problème concerne les choix de modélisation qui peuvent grandement influer sur la réussite de la vérification.

## Approximations probabilistes

Dans la section 2.4 du chapitre 2, nous avons vu qu'une fonction d'approximation est utilisée à chaque étape de complétion pour normaliser les transitions issues des paires critiques. Cette fonction est définie à la main par un expert et celui-ci la construit de telle sorte que la fonction d'approximation permette la convergence de la complétion vers un point fixe tout en évitant d'obtenir une approximation trop grossière (où il y aurait une intersection non-vide avec l'ensemble des termes indésirables). Cette définition se fait une seule fois et la fonction d'approximation ne change pas au cours de la complétion. Ainsi la fusion des états s'effectue toujours de la même façon, utilisant toujours le même ensemble d'états.

Toujours dans l'optique d'automatiser et de rendre plus efficace l'algorithme de vérification, une perspective serait de définir une fonction d'approximation (et par extension la fusion d'états) selon des heuristiques probabilistes. L'objectif de l'heuristique est de fournir une fonction d'approximation automatiquement et rapidement :

- permettant d'éviter une intersection non-vide entre l'approximation calculée et les propriétés tant que possible,
- de permettre à la complétion de terminer,
- et d'évoluer (ou non) à chaque étape de complétion.

Il est possible de jouer sur plusieurs paramètres lorsque l'on définit une fonction d'approximation. Lorsque l'on normalise une transition on doit choisir des états pour les nouvelles transitions produites. Ces états choisis peuvent être des nouveaux états ou des états existants. Parmi ces états existants, on peut en distinguer plusieurs types. Par exemple on peut les distinguer en fonction du langage qu'ils reconnaissent, ou encore du nombre de fois qu'ils ont déjà été utilisé pour la fusion d'états, ...

De plus il paraît nécessaire de limiter le nombre de nouveaux états utilisables pour les normalisations afin de permettre à la complétion de terminer.

Un exemple d'heuristique probabiliste possible serait de choisir au départ un ensemble fixe de nouveaux états qui seront utilisés pour la normalisation de transition. Ensuite à chaque étape de complétion le choix des états utilisés pour la normalisation peut se faire soit parmi les nouveaux états (probabilité de plus en plus faible, plus le nombre d'étape de complétion est grand) soit parmi des états déjà existants (probabilité de plus en plus forte, plus le nombre d'étape de complétion est grand). De plus ce choix parmi les états existants peut se faire en excluant, si possible, les états qui mèneraient — en prenant en compte les nouvelles transitions issues de la normalisation — à une intersection non-vide avec la reconnaissance d'un terme indésirable (ou un des ses sous-termes).

Étant donné qu'il existe un grand nombre de paramètres sur lesquels jouer, il existe un grand nombre d'heuristiques possibles. De plus, une heuristique efficace pour un problème ne l'est pas forcement pour un autre. Cependant il pourrait être intéressant de modéliser et implémenter un outil permettant à un utilisateur de jouer sur ces paramètres.

## Perspectives d'extensions de la complétion et du raffinement

Dans cette thèse nous avons vu que l'algorithme de complétion avait comme paramètre un automate d'arbres, un système de réécriture et une fonction d'approximation définie dans un langage particulier.

Dans le cadre de la transformation de documents XML, les automates des haies sont utilisés pour modéliser ces documents étant donné que les symboles ont des arités variables. Il pourrait donc être intéressant d'adapter la complétion d'automate pour les automates des haies.

De plus, dans [GR10], les auteurs utilisent des équations pour définir la fonction d'approximation, qui sont plus facile à définir pour l'utilisateur. Les équations permettent de définir des classes d'équivalences entre les états. Cependant, une mauvaise définition des équations peut entraîner le calcul d'une sur-approximation grossière. Il pourrait donc être intéressant d'adapter le raffinement d'approximation exposé dans cette thèse à la complétion avec équations.

Nous avons vu dans cette thèse une analyse en arrière qui est en fait une complétion exacte appliquée aux termes indésirables et utilisant un système de réécriture dont le sens des règles a été inversé. Le but étant de déterminer si l'intersection avec des ensembles obtenus aux étapes de complétion précédentes (calculées à partir des termes initiaux et du système de réécriture dont le sens des règles n'a pas été inversé) est vide ou non.

Il pourrait être intéressant de réaliser une analyse en arrière directement à partir des termes indésirables sans réaliser des étapes de complétion auparavant. C'est-à-dire, initialement nous avons des termes initiaux $I$, un système de réécriture $R$ et les termes indésirables $P$. Au lieu de calculer une approximation de $R^*(I)$, on calculerait une approximation (ou non, si possible) de l'ensemble $(R^{-1})^*(P)$ afin de déterminer si l'intersection $(R^{-1})^*(P) \cap I$ est vide ou non.

L'algorithme de complétion peut prendre du temps lors de son exécution, il pourrait donc être intéressant de trouver des solutions afin de réduire les temps de calcul. Par exemple, il est possible de paralléliser le processus de complétion afin de partager le calcul et réduire le temps d'éxécution. En effet à chaque étape de complétion il est possible de découper le système de réécriture et de compléter l'automate courant en utilisant chacun des systèmes de réécriture obtenus. Si à une étape de complétion $i$ on a l'automate $A_i$ et le système de réécriture $R$, on va découper $R$ tel que $R = R_1 \cup R_2 \cup \ldots \cup R_n$ et réaliser une nouvelle étape de complétion et obtenir un nouvel automate $A_{i+1}$ tel que $L(A_{i+1}) = L(C^{R_1}_{\gamma_1}(A_i)) \cup L(C^{R_2}_{\gamma_2}(A_i)) \cup \ldots \cup L(C^{R_i}_{\gamma_i}(A_i))$, où chaque $C^{R_j}_{\gamma_j}(A_i)$ peut être calculé sur un ordinateur différent et $\gamma_j$ sont des fonctions d'approximations $(1 \leq j \leq n)$.

Il faut noter que suivant comment le découpage est réalisé on peut obtenir des automates différents, de plus il est possible de jouer sur le nombre d'étape(s) de complétion réalisée(s) avec les systèmes de réécriture découpés afin d'obtenir d'autres résultats. Par exemple, toujours avec $A_i$ et $R = R_1 \cup R_2 \cup \ldots \cup R_n$, on peut réaliser une étape de complétion et obtenir un automate $A'_{i+1}$ tel que $L(A'_{i+1}) = L((C^{R_1}_{\gamma_1})^{(5)}(A_i)) \cup L((C^{R_2}_{\gamma_2})^{(5)}(A_i)) \cup \ldots \cup L((C^{R_i}_{\gamma_i})^{(5)}(A_i))$.

# Bibliographie


[ABB+05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[ACC07] A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. In *CSF*, pages 385–396, 2007.

[BBGL10] Yohan Boichut, Benoît Boyer, Thomas Genet, and Axel Legay. Fast Equational Abstraction Refinement for Regular Tree Model Checking. Technical report, July 2010.

[BBGM08] E. Balland, Y. Boichut, T. Genet, and P.-E. Moreau. Towards an efficient implementation of tree automata completion. In *AMAST*, pages 67–82, 2008.

[BCG+10] Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. The emptiness problem for tree automata with global constraints. In *LICS*, pages 263–272. IEEE Computer Society, 2010.

[BCHK08a] Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Finer is better : Abstraction refinement for rewriting approximations. In *Rewriting Techniques and Applications, RTA'08,*, Lecture Notes in Computer Science, 2008.

[BCHK08b] Yohan Boichut, Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Handling left-quadratic rules when completing tree automata. In *RP'08 proceedings, Workshop on Reachability Problems in Computational Models*, volume 223 of *ENTCS, Electronic Notes in Theoretical Computer Science*, pages 61–70. Elsevier, 2008.

[BCHK09] Yohan Boichut, Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Handling non left-linear rules when completing tree automata. *IJFCS, Intern. Journal of Foundations of Computer Science*, 20(5) :837–849, 2009.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Computer Aided Verification, CAV'96*, volume 1102, pages 1–12. Lecture Notes in Computer Science, 1996.

[BGJR07] Y. Boichut, Th. Genet, Th. Jensen, and L. Le Roux. Rewriting approximations for fast prototyping of static analyzers. In *Rewriting Techniques and Applications, RTA'07*, Lecture Notes in Computer Science 4533, pages 48–62. Springer, 2007.

[BGZ03] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Replication vs. recursive definitions in channel based calculi. In *ICALP*, pages 133–144, 2003.

[BH08a] Y. Boichut and P.-C. Héam. A Theoretical Limit for Safety Verification Techniques with Regular Fix-point Computations. *Information Processing Letters*, 2008. to appear.

[BH08b] Yohan Boichut and Pierre-Cyrille Héam. A theoretical limit for safety verification techniques with regular fix-point computations. *Inf. Process. Lett.*, 108(1) :1–2, 2008.

[BHK05] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic Verification of Security Protocols Using Approximations. Technical Report RR-5727, INRIA, 2005.

[BHK06] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Handling algebraic properties in automatic analysis of security protocols. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *International Colloquium on Theorical Aspects of Computing, ICTAC'06*, volume 4281 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2006.

[BHK08] Yohan Boichut, Pierre-Cyrille Héam, and Olga Kouchnarenko. Approximation-based tree regular model-checking. *Nord. J. Comput.*, 14(3) :216–241, 2008.

[BHK09] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Approximation-based tree regular model-checking. *Nordic Journal of Computing*, 2009. To appear.

[BHRV05] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In *proceedings of INFINITY*, number 4 in BRICS Notes Series, pages 15–24, 2005.

[BHRV06] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. *Electronic Notes in Theoretical Computer Science*, 149(1) :37–48, 2006.

[BHV04a] Ahmed Bouajjani, Peter Habermehl, and Tomas Vojnar. Abstract Regular Model Checking. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04) International Conference on Computer Aided Verification 2004*, LNCS, pages 372–386, Boston États-Unis, 2004. Springer Verlag.

[BHV04b] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Computer Aided Verification, CAV'00*, Lecture Notes in Computer Science 1855, pages 403–418. Springer-Verlag, 2000.

[BLP03] L. Bozga, Y. Lakhnech, and M. Perin. Pattern-based abstraction for verifying secrecy in protocols. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems,TACAS 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Boi06] Y. Boichut. *Approximations pour la vérification automatique de protocoles de sécurité*. Thèse de doctorat, Laboratoire Informatique de l'université de Franche-Comté, Université de Franche-Comté, Besançon, France, 2006.

[BT92] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *STACS*, pages 161–171, 1992.

[BW98] B. Boigelot and P. Wolper. Verifying systems with infinite but regular state spaces. In *Computer Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, jun 1998.

[CC05] H. Comon and V. Cortier. Tree automata with one memory set constraints and cryptographic protocols. *Theoretical Computer Science (TCS'05)*, 331, 2005.

[CCF+10] A. Caciula, Roméo Courbis, Violeta Felea, Pierre-Cyrille Héam, and R. Ionescu. Une approche parallèle et distribuée pour la complétion d'automates d'arbre. In *AFADL'10, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 43–46, Poitiers, France, June 2010. Papier court.

[CDL06] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14 :1–43, 2006.

[CDSV05] I. Cibrario, L. Durante, R. Sisto, and A. Valenzano. Automatic detection of attacks on cryptographic protocols : A case study. In Christopher Kruegel Klaus Julisch, editor, *Intrusion and Malware Detection and Vulnerability Assessment : Second International Conference*, volume 3548 of *Lecture Notes in Computer Science*, Vienna, 2005.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications : a practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 117–126, New York, NY, USA, 1983. ACM.

[CGJ+01] E. Clarke, S. Gruinberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. *Informatics, 10 Years Back, 10 Years Ahead, volume 2000 of Lecture Notes in Computer Science*, 2001.

[CHJK10] Roméo Courbis, Pierre-Cyrille Héam, Pierre Jourdan, and Olga Kouchnarenko. Approximations par réécriture pour deux problèmes indécidables. In *AFADL'10, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 7–10, Poitiers, France, June 2010. Papier court.

[CHK09] Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. TAGED approximations for temporal properties model-checking. In Sebastian Maneth, editor, *CIAA'09, 14th Int. Conf. and Application of Automata*, volume 5642 of *LNCS*, pages 135–144, Sydney, Australia, July 2009. Springer.

[Cla03] E. M. Clarke. Counterexample-guided abstraction refinement. In *TIME-ICTL*, page 7. IEEE Computer Society, 2003.

[Cou11] Roméo Courbis. Rewriting approximations for properties verification over ccs specifications. In *FSEN'11*, Teheran, Iran, 2011. to be publised.

[CS96] R. Cleaveland and S. Sims. The ncsu concurrency workbench. In *CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, pages 394–397, London, UK, 1996. Springer-Verlag.

[DAC98] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[DCC95] M. Dauchet, A.-C. Caron, and J.-L. Coquidé. Automata for reduction properties solving. *J. Symb. Comput.*, 20(2) :215–233, 1995.

[DNV90] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Proceedings of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes*, pages 407–419, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[EM07] S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Rewriting Techniques and Applications, RTA'07*, pages 153–168, 2007.

[FFGI94] N. De Francesco, A. Fantechi, S. Gnesi, and P. Inverardi. Model checking of non-finite state processes by finite approximations. In *In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95), Lecture Notes in Computer Science 1019*, pages 195–215. Springer, 1994.

[FGT04] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33(3-4) :341–383, 2004.

[FTT08] E. Filiot, J.-M. Talbot, and S. Tison. Tree automata with global constraints. In *Developments in Language Theory*, pages 314–326, 2008.

[FTT10] Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. Tree automata with global constraints. *Int. J. Found. Comput. Sci.*, 21(4) :571–596, 2010.

[Gen98] Th. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *proceedings of RTA*, volume 1379 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[GK00] Th. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Conference on Automated Deduction, CADE'00*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290. Springer-Verlag, 2000.

[GR10] Thomas Genet and Vlad Rusu. Equational approximations for tree automata completion. *J. Symb. Comput.*, 45 :574–597, May 2010.

[GT01] Th. Genet and V. Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. *Lecture Notes in Computer Science*, 2250, 2001.

[GTTVTT03] Th. Genet, Y.-M. Tang-Talpin, and V. Viet Triem Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *Proceedings of WITS'03*, 2003.

[GV98] P. Gyenizse and S. Vágvölgyi. Linear Generalized Semi-Monadic Rewrite Systems Effectively Preserve Recognizability. *Theoretical Computer Science*, 194(1-2) :87–122, 1998.

[GVTT01] Th. Genet and Valérie Viet Triem Tong. Reachability Analysis of Term Rewriting Systems with *timbuk*. In *proceedings of LPAR*, volume 2250 of *LNAI*, pages 691–702. Springer-Verlag, 2001.

[JKV09] F. Jacquemard, F. Klay, and C. Vacher. Rigid tree automata. In *LATA'09*, Lecture Notes in Computer Science, 2009. To appear.

[JKV11] Florent Jacquemard, Francis Klay, and Camille Vacher. Rigid tree automata and applications. *Inf. Comput.*, 209(3) :486–512, 2011.

[JRV06] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Tree automata with equality constraints modulo equational theories. In *IJCAR*, pages 557–571, 2006.

[KL07] W. Karianto and Ch. Löding. Unranked tree automata with sibling equalities and disequalities. In *ICALP*, pages 875–887, 2007.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3 :125–143, March 1977.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental Verification by Abstraction. In *proceedings of TACAS 2001*, Lecture Notes in Computer Science 2031. Springer-Verlag, 2001.

[Mes92] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1) :73–155, 1992.

[Mes07] J. Meseguer. The temporal logic of rewriting. Technical Report UIDCS-R-2007-2815, Dept of Computer Science, University of Illinois at Urbana-Champaign, September 2007.

[Mes08] J. Meseguer. The temporal logic of rewriting : A gentle introduction. *Concurrency, Graphs and Models : Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, pages 354–382, 2008.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[OT05] H. Ohsaki and T. Takai. ACTAS : A system design for associative and commutative tree automata theory. *Electronic Notes in Theoretical Computer Science*, 124(1) :97–111, 2005.

[Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57, 1977.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RJB94] Cleaveland R., Parrow J., and Steffen B. The concurrency workbench : A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15, 1994.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification (CAV '97)*. Springer-Verlag, 1997.

[SABL93] Ken Stevens, John Aldwinckle, Graham Birtwistle, and Ying Liu. Designing parallel specifications in ccs. In *In Proceedings of Canadian Conference on Electrical and Computer Engineering*, pages 983–986, 1993.

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking.* Vuibert, April 1999.

[SSMH04] H. Seidl, Th. Schwentick, A. Muscholl, and P. Habermehl. Counting in trees for free. In *ICALP*, pages 1136–1149, 2004.

[Tak04] T. Takai. A verification technique using term rewriting systems and abstract interpretation. In V. van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA-04*, Lecture Notes in Computer Science 3091, pages 119–133, Valencia, Spain, 3-5, 2004. Springer.

[TKS00] T. Takai, Y. Kaji, and H. Seki. Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In *proceedings of RTA*, volume 1833 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

## Résumé

Dans cette thèse nous nous intéressons à une technique de vérification basée sur les approximations par réécriture dans le but de l'automatiser et d'étendre son domaine d'application. L'utilisation de cette technique permet la vérification de propriétés pour des systèmes informatique. Une sur-approximation des termes atteignables par réécriture est calculée et nous pouvons déterminer si des termes indésirables (représentant la propriété) appartiennent à la sur-approximation. Si ils n'appartiennent pas à la sur-approximation alors on est sûr que les termes indésirables ne sont pas atteignables par le système et la propriété est vérifiée, sinon nous ne pouvons pas conclure à cause de la sur-approximation.

C'est dans ce cadre que se placent les contributions de cette thèse. Tout d'abord nous présentons une méthode de raffinement d'approximation, s'inspirant du paradigme CEGAR (Counterexample-Guided Abstraction Refinement), déterminant comment construire automatiquement une approximation qui ne contient pas de terme indésirable. Si cette construction échoue alors des termes indésirables sont atteignables et on peut conclure que le système analysé ne vérifie pas la propriété. De plus, à partir d'un graphe des réécritures, nous présentons comment vérifier trois modèles de propriétés LTL exprimant des conditions sur l'ordre d'application des règles de réécriture.

D'autre part, nous présentons dans cette thèse deux nouvelles applications de cette technique : la vérification de spécifications écrites avec l'algèbre de processus CCS (sans renommage) et l'analyse d'atteignabilité par approximation de deux problèmes indécidables pour les machines de Turing.



**Mots-clés :** approximation, régulier, réécriture, vérification, atteignabilité, raffinement